Domains and Denotational Semantics: History, Accomplishments and Open Problems

edited by
Achim Jung

with contributions by
Marcelo Fiore Achim Jung Eugenio Moggi Peter O'Hearn
Jon Riecke Giuseppe Rosolini Ian Stark

January 22, 1996

In this collection we try to give an overview of some selected topics in Domain Theory and Denotational Semantics. In doing so, we first survey the mathematical universes which have been used as semantic domains. The emphasis is on those ordered structures which have been introduced by Dana Scott in 1969 and which figure under the name *(Scott-) domains*. After surveying developments in the concrete theory of domains we describe two newer developments, the axiomatic and the synthetic approach. In the second part we look at three computational phenomena in detail, namely, sequential computation, polymorphism, and mutable state, and at the challenges that these pose for a mathematical model.

This presentation does by no means exhaust the various approaches to denotational semantics and it certainly does not describe all possible mathematical techniques which have been used to describe various aspects of programs. We hope that, nevertheless, it illustrates how a particular challenge (namely the modelling of recursive definitions) has given rise to an immensely rich theory, both in its general parts and in its applications.

Let us start with a few general remarks.

Denotational semantics has traditionally been described as the theory of true meanings for programs, or, to put it more poignantly, as the theory of *what* programs denote. In many cases, denotations have been built with the help of functions in some mathematical universe and so this position presupposes that the ontological status of sets and functions is firmly established. But there has always been an alternative viewpoint in which denotational semantics is seen as a *translation* from one formal system to another. This second position has become more and more popular over the last years, following the rapid progress on the programming language side which results in ever new and successful computational paradigms and which can hardly be ascribed merely to the desire to find syntactical descriptions of already existing mathematical objects.

However, the pragmatics of denotational semantics is essentially unaffected by the foundational stance one takes; the aims, hopes and concrete uses are the same. In highly condensed form, these may be described as follows: By translating from one formalism into another one expects to gain new insight into the object at hand. Elaborating slightly on this, we can say that the purpose of denotational semantics is threefold: to bring out subtle issues in language design, to derive new reasoning principles, and to develop an intuitive abstract model of the programming language under consideration so as to aid program development.

One would expect that the connection between programming language and mathematical model had to be very tight (as in the soundness and completeness theorems of mathematical logic) before the whole approach could be useful. Surprisingly enough, this is not the case. A true one-to-one correspondence (called *full abstraction* in the jargon) has seldomly been achieved, yet the discovery and the transfer of reasoning principles has indeed taken place. A most convincing example of this is the so-called *context lemma* for functional languages [Mil77]: the mere existence of a model consisting of functions which is loosely connected with the language ("adequacy") allows one to infer in a series of very simple steps that the equivalence of two terms depends only on the behaviour under application to arguments (as is the case with mathematical functions).

Still, a closer correspondence between Syntax and Semantics ought to result in better applications for the theory. As this text shows, much remains to be done and many fascinating riddles remain.

Classical Domain Theory

Achim Jung

Domain theory started in 1969 when Dana Scott explored the possibility of using ordered topological spaces to give meaning to first typed and then untyped λ-calculi [Sco93, Sco72]. 25 years may be too short a period to warrant the attribute "classical" but these 25 years have seen a tremendous development of this theory and its incorporation among the basic concepts of theoretical computer science (on par with recursion theory, complexity theory and formal languages) so that the expression is well justified.

Here we also use "classical" to distinguish between the concrete set-theoretic structures known as *directed-complete partial orders* (or *dcpo*'s for short) from the more abstract categorical approaches described in the next two sections.

The category of dcpo's and Scott-continuous functions serves as a convenient ambient universe in which one may study more refined notions. Foremost among these is the concept of *approximation* expressed via continuous and algebraic domains. The effect of adopting an axiom of approximation is that each domain may be seen as a completion of a countable, possibly even decidable, structure. The behaviour of ideal elements is completely determined by the behaviour of its approximants. In contrast, the ambient category DCPO is rather awkward to work with when it comes to concrete calculations.

Within approximated domains, that is, within the categories CONT and ALG we may look for additional structure to model computational phenomena. Higher types, for example, require cartesian closure. Neither CONT nor ALG is cartesian closed but from [Smy83a] and [Jun90] we know essentially all cartesian closed full subcategories. Among the maximal ones are FS-domains (in the continuous case) and bifinite domains (in the algebraic case), see [AJ94b, Chapter 4].

Recursive types pose no problem for ALG, CONT, nor for any of its cartesian closed subcategories. They are resolved by the fundamental technique of bilimits [AJ94b, Chapter 5], devised early on by Scott [Sco72]. Adding algebraic structure in a free fashion is also possible [AJ94b, Section 6.1] but the general theorem applies to ALG and CONT only. It is an open problem whether there is a characterization of those algebraic theories which yield free algebras within one of the cartesian closed subcategories. Even the probabilistic powerdomain construction stays within CONT as was shown in [Jon90] but again it is unknown if it stays within FS-domains. Hence we may have to accept the fact that there is no single category of domains which meets all needs.

In his attempts to capture sequentiality in a mathematical model, Gerard Berry

[Ber78, Ber79] developed an alternative domain theory based on the notion of *stability*, seemingly incompatible with Scott-continuity (see [Gun92, Section 5.2] for an exposition). Through the very recent work of François Lamarche [Lam] and Mathias Kegelmann [Keg95] we now see that both Scott-continuity and stability are manifestations of a single concept, that of a factorized domain, where the information order can be decomposed uniquely into two suborders. It is a fascinating question to see how this factorization reflects different orders between program terms.

A further development of the last 10 years is the clarification of the connections between denotational semantics and program logics. Initiated by Michael Smyth [Smy83b] (another precursor is [CDCHL84]) and carried much further by Samson Abramsky [Abr87, Abr91b], we now have a fully developed logical theory of domains in place. The fundamental insight is that open sets may be seen as "observable" properties and that Stone-duality [GHK$^+$80, Joh82] is the connection between domain and logic. This theory has been brought to bear on functional languages [AO93] and communicating systems [Abr91a] by its inventors and has been applied by many others since. Domain logic is propositional in character and the obvious question of how to embed it into a richer framework has been frequently asked but not been answered satisfactorily. Synthetic domain theory is one of the more coherent projects to tackle this problem.

One of the earliest successes of domain theory, as already mentioned, was the solution of recursive type definitions which involve positive and negative occurrences of variables, such as in $D \cong [D \to D]$. While this was happily accepted as a technique to construct meanings of programs, only very recently has it been clarified, in which sense these solutions are canonical. Peter Freyd gave a categorical condition for what it means in Computer Science to have a canonical solution to a recursive type definition, namely, that initial algebra and final coalgebra of the corresponding functor coincide [Fre91, Fre92]. From this (for covariant functors) he derived a condition for functors of mixed variance. Building on this, Andrew Pitts developed it into an induction-coinduction principle for the elements of canonical solutions, see [Pit94, Pit96].

While ordered sets arise naturally in the theory of computable functions (since one partial recursive function may extend another one by converging on more inputs and coinciding with the former whenever that is defined), it has always been emphasized by Scott that the information order in domains may also be interpreted as giving approximations to a space of ideal elements on the top of the domain. To put it differently, in recursion theory and in denotational semantics all points in the domain are equally needed as meanings (even the concepts of *total* function and maximal element do not coincide; a much more elaborate theory is needed [Ber93]), whereas under the second interpretation everything except maximal points is considered auxiliary. This viewpoint has been developed in two areas recently, in database semantics and in measure theory. In the former one enriches a space of values (as in database theory) with partial elements so as to allow partial information and complicated types to be incorporated. See [BJO91, JP95, Puh95] for the current state of this approach. Much remains to be done in this field. The second application was developed almost single-handedly by Abbas Edalat over the last few years [Eda95b, Eda95a]. He showed that when a topological Hausdorff space is embedded as singleton sets in its "upper space" of compact subsets, then the Borel measures on the space are embedded as maximal elements in the probabilistic powerdomain of the upper space. By this embedding it is possible to approximate Borel measures from below, hence giving an order theoretic account of approximation of measures. In contrast, classical measure theory always had to invoke various topologies on the space of measures to express convergence. Furthermore, the probabilistic powerdomain contains completely new "measures" and in many circumstances allows the construction of an increasing chain of approximating measures to a desired Borel measure. Edalat has demonstrated the applicability of his approach in areas as varied as dynamical systems, neural networks and image compression. This connection between domain theory and mainstream mathematics remains a fascinating and active area of research.

Axiomatic Domain Theory

Marcelo P. Fiore

The *denotational semantics* approach to the semantics of programming languages interprets the language constructions by assigning elements of mathematical structures to them. The structures form so-called *categories of domains* and the study of their closure properties is the subject of *domain theory* [Sco70, Sco82, Plo81, GS90, AJ94b].

Typically, categories of domains consist of suitably complete partially ordered sets together with continuous maps. But what is a category of domains? The main aim of *axiomatic domain theory* is to answer this question by axiomatising the structure needed on a mathematical universe so that it can be considered a category of domains. Criteria required from categories of domains can be of the most varied sort. For example, we could ask them to

- have a rich collection of type constructors: sums, products, exponentials, powerdomains, dependent types, polymorphic types, etc;
- have fixed-point operators for programs and type constructors;
- have only computable maps [Sco76, Smy77, Mul81, McC84, Ros86, Pho91, Lon95];
- have a Stone dual providing a logic of observable properties [Abr87, Vic89, Zha91].

An additional aim of the axiomatic approach is to relate these *mathematical* criteria with *computational* criteria.

As we indicate below an axiomatic treatment of various of the above aspects is now available but much research remains to be done.

Developments

In the beginning, the axiomatic treatment of domain theory was scattered; it concentrated on fixed-points, mainly of endofunctors but also of endomorphisms.

Concerning fixed-points for endofunctors, already in [Sco72], Scott mentions a suggestion by Lawvere aiming at providing a categorical framework for performing the D_∞ construction. But it was not until [Wan79] that the solution of recursive type equations in categories of domains was first treated *abstractly*, in the sense that no commitment to a particular category of domains was required. Subsequently this approach was developed in [SP82]. The approach was very much appreciated as a *unification* of the techniques for solving recursive type equations in categories of domains, but its *axiomatic* character remained overlooked. For instance, it lead Lehmann and Smyth [LS81] to outline the first abstract setting for specifying both algebraic (in the ADJ jargon) and recursive types, but these ideas were not pursued further. In [Fre90], aiming at an axiomatic treatment of recursive types, Freyd revisited the previous approaches. And, in [Fre91, Fre92], he proposed a *universal* approach (in the category-theoretic sense) for solving recursive type equations. There he introduced *algebraically compact* categories and established their fundamental property: that bifunctors on them have canonical and minimal fixed-points.

This has been a first important step towards an *axiomatic theory of recursive types* (see [Sim92] and [Fio94a, Chapters 6–8]). Other work on algebraic compactness can be found in [Adá93, Bar92].

Concerning fixed-points of endomorphisms, it was noticed by [HP90], after studying the work of [Law64, Law69], that in the presence of cartesian closure they are inconsistent with coproducts (empty or binary). Also algebraic compactness (which yields zero objects) is inconsistent with cartesian closure. This, in principle, precludes a unified treatment of sums, products, exponentials and recursive types via the usual universal properties. However, it was via a direct semantic analysis of non-terminating computations [Plo85] involving *categories of partial maps* [RR88] and, in particular, via the notion of *partial cartesian closure* [LM84] that an appropriate categorical setting emerged.

With this background it was possible, for the first time, to consider categorical models for a rich type theory with recursive types. In [Fio94a, FP94], a notion of categorical model for the metalanguage FPC —a type theory with sums, products, exponentials and recursive types [Plo85, Gun92, Win93]— was defined. Very roughly, categorical models of FPC are algebraically compact partial cartesian closed categories with binary coproducts.

Impact of axiomatic domain theory

In relating operational and denotational semantics. The investigation of the relation between operational and denotational semantics started with a question of Scott in [Sco69]. An answer for PCF (a higher-order functional programming language with fixed-point operators, and base type and arithmetic operations for natural numbers) was given by Plotkin in [Plo77]; there he proved the *soundness* and *adequacy* of the standard semantics of PCF for a call-by-name evaluator. A computational soundness and adequacy result is a correspondence theorem between *operational termination* and *denotational existence*; it generally states that a program terminates according to the operational semantics if and only if its denotational semantics denotes a value. The first axiomatic version of such a result was provided by Berry in [Ber79] (see also [BCL85]) for PCF with respect to a class of models including both the standard one and the *stable* one. In this vein, [Bra94] considered a term language for intuitionistic propositional linear logic extended with fixed-point operators and provided a computational soundness and adequacy result with respect to a categorical semantics.

Concerning languages with recursive types, in [Plo85], FPC was considered as a programming language with a call-by-value operational semantics and a denotational semantics in **pCpo** (the category of cpos and partial continuous functions) and a computational soundness and adequacy result was proved. (Related results can be found in [ML83, MC88].) In [Fio94a], the author gave an axiomatic version of Plotkin's proof for an axiomatisation of absolute non-trivial domain-theoretic models of FPC. From this result it follows that not only the standard model **pCpo** provides a computationally sound and adequate interpretation for FPC but so also do many full subcategories of domains and functor categories over **pCpo**.

In reasoning principles for recursive types. The universal approach taken by the category-theoretic solution of recursive type equations provides, as a by-product, reasoning principles for recursive types. Lehmann and Smyth, and Plotkin [LS81, Plo81], exploited the initiality property of the solution of covariant recursive types (like lists, trees, etc.) to formulate an abstract *induction principle* generalising that of the natural numbers. Motivated by the introduction of algebraic compactness, [Smy91] studied the dual reasoning principle, *coinduction*, in some particular cases. In [Pit94, Pit93], Pitts pursued this line of research further. Drawing upon the initiality/finality universal property given by the compactness axiom, he established a mixed induction/coinduction property of abstract relations on recursive domains in $\mathbf{Cppo}_\perp$ (the category of pointed cpos and strict continuous functions). Abstract category-theoretic accounts of these issues can be found in [Fio93, HJ95].

In type theory. In [CP92], Crole and Pitts introduced a higher-order typed predicate logic for fixed-point computations. This was done by exploiting Moggi's treatment of computations using monads [Mog91], and by introducing the key notion of *fixpoint object*. Fixpoint objects were partly inspired by Martin-Löf's non-standard "iteration type" [ML83], and give a categorical characterisation of general recursion at higher types similar to the characterisation of primitive recursion at higher types in terms of Lawvere's concept of *natural number object* [LS86].

A type-theoretic approach to domain theory is that of [Plo93]. There, rather than considering directly possible categorical structure, the idea is to work within a type theory pursuing the analogies: intuitionistic exponential = function space, and linear exponential = strict function space. More precisely, the basic setting is that of second-order intuitionistic linear type theory enriched with a fixed-point operator for endomorphisms. Then, in the presence of a modified form of Reynold's parametricity the category of linear maps is shown to be algebraically compact with respect to definable endofunctors.

In [Mog95], Moggi describes monadic and incremental approaches to denotational semantics. There, incorporating ideas from axiomatic and synthetic domain theory, typed metalanguages are used to capture relevant structure of semantic categories. Then, by translation into the metalanguage, a variety of programming languages can be interpreted.

In models of domain theory. New non-order-theoretic models of domain theory have been found in connection with research towards establishing a *representation theory* for domains (see '§ Representation theory for domains' below). For illustrative purposes we present one such model. Domains are spaces equipped with a notion of *approximation* (the information order) and a notion of *passage-to-the-limit* (the sup operator). We now consider objects with an *algebraic* (rather than universal) notion of passage-to-the-limit. To this purpose we introduce a notion of *formal-sup operator* due to the author and Gordon Plotkin. Let $\mathcal{D}$ be a cartesian closed category (think of **Poset**, the category of posets and monotone functions), let w be an object in $\mathcal{D}$ (think of ω, the first infinite ordinal), and let $s : w \to w$ be a morphism in $\mathcal{D}$ (think of succ, the successor function). A *formal-sup operator* (for diagrams of shape w under the invariance s) on an object D in $\mathcal{D}$ is given by a map $\bigvee : D^w \to D$ satisfying the following three algebraic laws:

(Constant)
$$\begin{tikzcd} D \arrow[r, "\lambda(\pi_1)"] \arrow[dr, "\mathrm{id}_D"'] & D^w \arrow[d, "\bigvee"] \\ & D \end{tikzcd} \qquad \big(\text{i.e. } \textstyle\bigvee_n \langle x \rangle = x\big),$$

(Diagonal)
$$\begin{tikzcd} D^{w \times w} \cong (D^w)^w \arrow[r, "\bigvee^w"] \arrow[d, "D^\Delta"'] & D^w \arrow[d, "\bigvee"] \\ D^w \arrow[r, "\bigvee"'] & D \end{tikzcd} \qquad \big(\text{i.e. } \textstyle\bigvee_n \langle \bigvee_m \langle x_{m,n} \rangle \rangle = \bigvee_n \langle x_{n,n} \rangle\big),$$

$$\text{(Shift)} \qquad \begin{tikzcd} D^w \arrow[r, "D^s"] \arrow[dr, "\bigvee"'] & D^w \arrow[d, "\bigvee"] \\ & D \end{tikzcd} \qquad \big(\text{i.e. } \bigvee_n \langle x_n \rangle = \bigvee_n \langle x_{s(n)} \rangle \big).$$

A *formal cpo* is an object equipped with a formal-sup operator. For formal cpos $(P, \bigvee_P)$ and $(Q, \bigvee_Q)$, a map $f : P \to Q$ in $\mathcal{D}$ is said to be *continuous* if it satisfies the following law:

$$\text{(Continuity)} \qquad \begin{tikzcd} P^w \arrow[r, "f^w"] \arrow[d, "\bigvee_P"'] & Q^w \arrow[d, "\bigvee_Q"] \\ P \arrow[r, "f"'] & Q \end{tikzcd} \qquad \big(\text{i.e. } f(\bigvee_P \langle x_n \rangle) = \bigvee_Q \langle f(x_n) \rangle \big).$$

We write $\bigvee_{(w,s)} \mathcal{D}$ for the category of formal cpos and continuous maps. The reader can readily check that our running example $\bigvee_{(\omega,\mathrm{succ})}\mathbf{Poset}$ is **Cpo**, the category of cpos and continuous functions. What happens if we replace **Poset** by **Preo** (the category of preorders and monotone functions)? Nothing, $\bigvee_{(\omega,\mathrm{succ})}\mathbf{Preo}$ is again **Cpo**. The surprise comes when we consider the construction for structures with not only a one-dimensional notion of approximation (as the above examples) but with *higher-dimensional* notions of approximation. For instance, consider $\mathbf{Poset}_\wedge$, the category of posets with pullbacks (here the pullback "squares" provide a "two-dimensional" notion of approximation) and stable functions. Then, $\bigvee_{(\omega,\mathrm{succ})}\mathbf{Poset}_\wedge$ cannot be enriched over **Cpo** in a relevant sense [Fio94b] but in it the constructions of domain theory (as, for example, the existence of uniform fixed-point operators and the solution of recursive domain equations) are possible (see '§ Representation theory for domains' below). The above formal-cpo construction has been studied in generality by the author, Plotkin and Power.

Directions

Representation theory for domains. A *representation theorem* is a result that classifies the models of mathematical structures in terms of more concrete models; allowing the study of the general through the study of the particular. (E.g. one such result is Cayley's theorem for groups stating that every group is isomorphic to a subgroup of a group of permutations.) The purpose of setting up a representation theory for domains is to understand the extent to which axiomatisations constrain their models. Work in this direction can be found in [Fio94b]. There the author provided a strong axiomatisation for which enrichment and representation theorems in **Cpo** were proved. Corresponding enrichment and representation theorems for weaker axiomatisations are being explored by the author, Plotkin and Power. This has uncovered a new range of models (among which $\bigvee_{(\omega,\mathrm{succ})}\mathbf{Poset}_\wedge$) based on higher-dimensional *geometric* structures. In fact, these models yield *models of domain theory* in the sense of [Plo95]. According to Plotkin, these consist of a monoidal adjunction between a cartesian closed category and a symmetric monoidal closed one (i.e. a model of intuitionistic linear type theory [BBHdP93]) together with natural axioms for recursion; and, as he shows, they admit the standard techniques for solving recursive domain equations via a version of the limit/colimit coincidence theorem.

Axiomatic and synthetic domain theory. Axiomatic domain theory and synthetic domain theory [Mul81, Ros86, Hyl91, Pho91, Tay91, RS94, Lon95, Ros95] are complementary approaches. On the one hand, synthetic domain theory tries to identify domains, complying with the requirements of axiomatic domain theory, within a universe of sets. On the other hand, as we exemplify below, it is conceivable that one could embed models of axiomatic domain theory in a universe of sets, along the lines prescribed by synthetic domain theory. For example, building upon [Fio94b], for a strong axiomatisation, the author has obtained a representation theorem of the form: every small model has a full and faithful representation in a model of cpos and continuous functions in a presheaf topos; furthermore, the representing model is a full reflective subcategory of the topos. Analogous embeddings for weaker axiomatisations are under investigation by the author, Plotkin and Power.

Type theory. An important direction of research is the formalisation of semantic developments in logical frameworks [dB80, CAB$^+$86, CH88, HHP92, ACN90] for their subsequent use in machine-assisted reasoning about programs. To this purpose it might be helpful to close the gap between the language of category theory and that of type theory. Recent work in program verification in synthetic domain theory using the LEGO proof checker [Pol95] can be found in [Reu95].

Relating models of FPC. It is an interesting and rather straightforward observation that two interpretations of the simply typed λ-calculus in a cartesian closed category, for which the base types get isomorphic objects, are essentially the same; in that the interpretations of types are canonically natural isomorphic and the interpretations of terms are interdefinable (via the canonical natural isomorphisms). A corresponding result (and generalisations involving interpretations in different models) for FPC would be worth investigating. For instance, this question is relevant to abstract proofs of adequacy (see [Fio94a, Section 10.3]). The main difficulty here is that in the presence of recursive types one has to overcome a kind of *coherence* problem.

Notions of computation. In [Fio94a] an axiomatic setting for partiality was developed. A similar attempt for other notions of computation has not been pursued. A natural step would be to consider non-determinism. One may regard non-determinism as a free construction [HP79] and hence as a notion of computation in the sense of Moggi [Mog91]; alternatively one may develop an abstract setting of observable properties in the vein of [Abr87, Rob88] (see also [Vic89, Joh82]).

Polymorphism and recursion. It would be interesting to develop axiomatic frameworks for polymorphism with recursion (accommodating both domain-theoretic models [CGW89] and per models [AP90, FMRS90]) together with a corresponding representation theory.

Game semantics. The application of *game theory* to semantics has provided new insights and results [AMJ94, HO95b, AM95]. In particular, [AMJ94] and [HO95a] have constructed *intensionally fully abstract semantic models* of PCF which yield fully abstract models by extensional collapse. As advocated by Abramsky, axiomatic studies of game semantics may lead to abstract full abstraction results (as had happened with adequacy results).

Acknowledgements

I thank P. Cenciarelli and G. Plotkin for their comments.

Synthetic Domain Theory

Eugenio Moggi and Giuseppe Rosolini

Background and motivations

In the second part of the seventies Dana Scott suggested that domains for denotational semantics could be nicely embedded into a model of intuitionistic set theory (*i.e.* an elementary topos, see [Joh77]) so that domains would be "sets" with some *very* peculiar properties and all functions between them continuous. The standard theory for domains, which is required for denotational semantics, would then have to follow from set-theoretic principles.

The main goal of Synthetic Domain Theory is to bridge the gap between categories of domains, which provide adequate models for programming languages, and set-theoretic universes, where type-theory and logic are interpreted. In this way one would like to obtain a viable framework for the practice of Denotational Semantics and Program logics as originally envisaged by Dana Scott.

An important step in this direction is to develop axiomatisations of categories of domains (this is the main goal of Axiomatic Domain Theory). These axiomatisations should be consistent with known categories of domains, and should also provide structural requirements for new semantic categories.

History and state of the art

The early work was by Robin Grayson, Martin Hyland, David McCarty, Phil Mulry and Giuseppe Rosolini (see [Hyl82, McC84, Mul81, Ros86]). It focused on the effective topos *Eff* based on Kleene's realizability, and it was noted that effectively given Scott domains form a full sub-category of *Eff*, and that there is an object Σ of "recursively enumerable" truth values, which can be defined in the *internal logic* as

$$\Sigma = \{p \in \Omega | \exists f{:} N \to N.p \Leftrightarrow (\exists n \in N.f(n) = 0)\}$$

such that the r.e. subsets of N are in a one-one correspondence with the maps $N \to \Sigma$.

Wesley Phoa pointed out more interesting, peculiar properties of Σ [Pho91, Pho90]. The order on Σ is defined in terms of Σ-paths:

$$p \leq q \Leftrightarrow \exists \alpha{:} \Sigma \to \Sigma.(p = \alpha(\bot) \wedge q = \alpha(\top)).$$

Martin Hyland presented the first set of properties for an object Σ of an elementary topos necessary to develop domain theory as the theory of the *replete* objects (see [Hyl91, HM95]): an object X is *replete* if, whenever $\Sigma^f{:} \Sigma^B \to \Sigma^A$ is iso, then X has the unique extension property with respect to f, *i.e.* for every $\alpha{:} A \to X$ there is a unique $\beta{:} B \to X$ such that $\beta \circ f = \alpha$.

Roughly at the same time, Peter Freyd discovered the universal properties of the solutions of domain equations clearing the way for their full category-theoretic treatment (see [Fre91, Fre92]). This discovery spurred research toward an axiomatic presentation of categories of domains (see [Sim92, Fio94a, FP94]), which encompassed that centered on O-categories. Freyd's axiomatic presentation sets an important criterion about properties of functors in a model of SDT (see [Hyl91, Ros95]).

By pursuing the SDT approach in the setting of realizability toposes it is easy to model both polymorphism and recursive definitions. The first such model was based on *complete extensional PERs* (see [FMRS90, Ros92]). More recently, John Longley investigated a variety of realizability models for SDT (see [Lon95]). Some of these models do not satisfy the axioms for Σ proposed by Hyland and others. However, Longley and Simpson have proposed weaker axioms for Σ and a different category of *domains*: the well-complete objects, which include the replete ones.

Directions

The research directions can be classified under two main headings: study of specific models, further development of the axiomatic setting.

Realizability models. Only few realizability models have been studied in some depth (mainly by Hyland, Rosolini, Phoa, Streicher, Longley). Preliminary studies have shown that realizability models can give very different insights into SDT and type theory, simply changing the partial applicative structure or modifying the notion of realizability (see [Pho94, Lon95, Reu95]), this has no parallel in sheaf models for SDT.

Moreover, one could also broaden the current realizability framework to encompass typed versions of realizability, which should be directly applicable to categories of domains used in denotational semantics.

Axiomatic and synthetic domain theory. It is important to identify a class of models of SDT, in which a wide range of categories of domains (identified axiomatically) can be *embedded*. These models (together with the realizability models) could provide the intended models for expressive type theories, where programming languages can coexist with set-theoretic reasoning. Three main steps seem necessary.

- To identify axiomatic descriptions (along the lines proposed by Freyd, Plotkin and others) for categories of domains, including the categories used in denotational semantics. These descriptions should represent minimal requirements on structure and properties of semantic categories.
- To investigate models of SDT, *i.e.* set-theoretic universes together with a natural construction of a good sub-category which complies with the axiomatic descriptions above. Only the replete construction has been investigated in some generality (see also [HM95]). Other promising constructions like the well-complete objects introduced by Longley and Simpson, and the formal cpos introduced by Fiore, Plotkin and Power deserve further investigation. In some cases these constructions do not rely on all properties of a topos, *e.g.* the replete construction can be performed in any monoidal closed category. It is important to apply such constructions to familiar categories of domains, and test whether they give rise to a proper subcategory, *e.g.* it is not known what are the replete objects in the category of cpos.
- To investigate general ways of embedding categories of domains (*e.g.* cpos), complying with the axiomatic descriptions, into SDT models (*e.g.* based on sheaf toposes).

Another important issue is how much of classical domain theory can be *recovered* from the axioms for SDT, this can be approached in different ways.

- One could express the concepts of classical domain theory (e.g. various combinators and domain constructions, admissible subsets, finite elements) in the language of SDT, and then try to derive from the axioms of SDT most of the classical results (see [RS94, Reu95]).
- One could try first to rethink classical domain theory more abstractly (e.g. powerdomains as free algebras constructions, Freyd's axiomatic presentation, duality in terms of a dualising object), and look for the SDT analogue at this more abstract level (see [TP90, Tay95]).

Finally, one should also investigate axiomatisations more *usable* for the practice of program verification, though derivable from more *fundamental* axioms of SDT.

PCF and the Problem of Full Abstraction

Jon G. Riecke

History

PCF (Programming Computable Functions) is a spare, purely functional language originally defined by Dana Scott in 1969 as the term language of the logic LCF (Logic of Computable Functions) (see [Sco93]). Scott's main purpose in explicating LCF was to show how a simple model based on lattices and continuous functions, with a least fixed point interpretation of recursion, could be used in deriving a program logic. As a final remark, Scott mentioned a curious function in the model that seemed not to be definable in PCF:

$$por(d,e) = \begin{cases} \mathit{true} & \text{if } d \text{ or } e = \mathit{true} \\ \mathit{false} & \text{if } d = e = \mathit{false} \\ \bot & \text{otherwise} \end{cases}$$

In 1977, Plotkin demonstrated the importance of the remark. He proved that not only was the "parallel or" function not definable, but it caused two terms to be distinguished that cannot be distinguished in PCF [Plo77]. For instance, given the term F defined

```
λx:bool. λf:bool→bool→bool.
   if (and (f true diverge)
           (f diverge true)
           (not (f false false)))
      then x
      else true
```

where `diverge` encodes any infinite loop of type `bool`, (F `true`) and (F `false`) must be different in the model, since when applied to *por*, the meaning of the former returns *true* whereas the meaning of the latter returns *false*. However, in PCF, there is no term which will cause all three parts of the `and` in `F` to return `true`. More generally, placing (F `true`) or (F `false`) in any context—i.e., a term with a hole in them—will produce exactly the same answer. That is, the terms are "operationally equivalent" [Mor68]. When operational and denotational equivalence do coincide, the model is called *fully abstract*. Plotkin went on to establish that the model based on cpo's (Scott's model construction generated from base types without top) and continuous functions *is* fully abstract for PCF enhanced with a "parallel if" operation [Plo77]. Stoughton [Sto91] later showed that adding a term for *por* was enough.

But what about the original problem—can one describe the sequential functional computation of PCF via some abstract denotational model? The problem has fascinated semanticists for over 25 years, and is important for at least three reasons. First, the problem is *robust*. For instance, if we build Scott-style models for PCF with only booleans or only naturals, for call-by-value versions of PCF, or for languages like FPC with recursive types or polymorphic λ-calculus with recursion, the same problem arises: the model contains parallel elements that cause unwanted distinctions. Second, deterministic sequential computation appears to be *fundamental*, especially when we move from purely functional languages to languages with state. In naïve Scott-style models for languages with state, a determinate *por* also arises in the model, but this form of determinate parallelism requires copying the state; keeping one state and parallelism forces us to nondeterminism. In essence, parallelism in a language with state seems to impose a choice between nondeterminism or efficiency. The naïve models thus prevent one from deriving helpful principles about determinate, stateful programs—surely an important class of programs.

Third, the sequentiality problem is *embarrassing*. One of the key goals of semantics is to describe and formalize the structure of computation. There can be no doubt that *por* is computable in Turing's sense: one can simply run two arguments via time-slicing, returning `true` if one halts at `true` and returning `false` if both halt at `false`. To the semantics novice, this seems to expose a flaw in the proof that *por* is not definable: PCF is Turing complete, so surely *por* can be defined. But this can only be done by encoding entire computations of type `bool` as numbers; the *por* function cannot be computed in isolation. In other words, if we try to describe the structure of programs without destroying that structure, sequentiality is one of the first problems one encounters, and shows just how limited our knowledge is of semantics.

The sequentiality problem, nevertheless, is vague as stated. What does it mean to capture the idea of sequentiality in an abstract way? One could say the problem is simply to build a fully abstract model of PCF, but the unilluminating term model built from the operational equivalence relation would satisfy that goal. One could demand that the model be constructed out of partially-ordered sets and continuous functions, so that Scott's interpretation of recursion as least fixed point is possible. The basic term model does not have enough limit points, so it fails that criterion. But the following theorem due to Milner [Mil77] codifies the weakness of this criterion.

Theorem 1 (Milner, 1977) *There is exactly one continuous, inequationally fully abstract model for PCF.*

A model is *inequationally fully abstract* if it is based on partial orders, and if denotational approximation coincides with operational approximation, where M operationally approximates N if in any context $C[\cdot]$ such that $C[M]$ and $C[N]$ have base type, if $C[M]$ reduces to a final answer, then $C[N]$ returns the same final answer. (See [Sto90] for a discussion of *equationally* fully abstract models). Milner constructed the model using a sophisticated inverse limit, where the finite models were built from operational equivalence classes of terms. Nevertheless, it still tells us little independent of the operational equivalence relation. The sequentiality problem boils down to finding interesting, illuminating characterizations of Milner's model.

Developments

Berry, Curien, and Levy's excellent article [BCL85] gives a summary of approaches to the sequentiality problem before 1985; this article covers the main developments since then. Basically, there have been five (somewhat overlapping) attempts to describe Milner's model: term models, domain-theoretic models, sequential algorithms models, games models, and logical relation models.

Term models form the first class. Milner's construction was followed by two more distinct constructions. Mulmuley's model [Mul87] takes the original *lattice* model of Scott and, using syntactic closures (retractions that are greater than the identity), collapses it syntactically to the fully abstract model. The offending parallel elements are sent to the top elements of the lattices, which are then eliminated. Stoughton's model [Sto88] starts with the inductively reachable subalgebra, the set of elements of Plotkin's cpo model which are lubs of definable elements, and uses a syntactically defined preorder to reach Milner's model. The construction was improved by Jung and Stoughton, who used a syntactically defined *projection* to reach Milner's model [JS93]. The technical advantages and disadvantages to each construction are described succinctly in [JS93].

The second approach, *domain-theoretic*, comes from Berry's notion of *stable* functions, a subset of the continuous functions on dI-domains that does not include *por* [Ber78]. The stable model is quite interesting, but not fully abstract; in fact, even though it seems closer at first-order type to the fully abstract model, the inequational theory is incomparable to that of Plotkin's model [JM91]. Bucciarelli and Ehrhard [BE91, Ehr93] refined the model with stronger stability conditions on functions to arrive at a model that is fully abstract for the first-order fragment of PCF. Brookes and Geva [BG93] also achieve full abstraction for a fragment of PCF using domain-theoretic ideas. Recently, Winskel [Win94] refined Berry's bidomains to bistructures, and arrived at a better model incorporating stable and pointwise orders.

The initial failures of stable functions led to the third approach of *sequential algorithms*. Previously, Berry and Curien [BC85, Cur86] described a model composed of algorithms (not functions) for PCF which had a game-like structure. Computations proceed by dialogues of questions and answers composed of "filling cells". Cartwright and Felleisen, subsequently built a fully abstract model for SPCF—an extension of PCF that remains sequential but includes errors and a simple control operator—out of question/answer trees [CF92]. Curien pointed out that they had defined on a subtly different version of sequential algorithms [Cur92, CCF94]. While this result is not for PCF itself, the model is quite interesting in many respects.

A fourth approach that is closely related to sequential algorithms is *game semantics*. Games-based models are almost like sequential algorithms, except that the same question may be repeated in computations. The first results, due to Abramsky and Jagadeesan used game semantics to achieve full completeness for a fragment of linear logic [AJ94a]; "full completeness" means that any element of the model (in this case, a strategy) corresponds to a proof. The connection with full abstraction should be clear: if all elements of the model can be represented syntactically, then the model is almost assuredly fully abstract. Subsequently, three groups (almost simultaneously) found ways to extend the results to PCF. Abramsky, Jagadeesan, and Malacaria used a certain class of "history-free strategies" where answers are provided only to the previous question, and where new questions cannot use the entire list of previous moves [AMJ94]. These games can be used to describe directly the ! operation of linear logic. Due to the direct use of !, two quotients are required to arrive at Milner's model: the first to eliminate distinctions between playing in different copies of !, and the second to eliminate distinctions based on order of evaluation, etc. Hyland and Ong [HO] and Nickau [Nic94] described games for PCF without interpreting ! directly. In their "dialogue games", every answer must have a unique justifying question, and where answers are supplied to only immediately proceeding questions. Their construction requires only the last quotienting step of the history-free games, although composition seems more difficult to define.

Sieber [Sie92] pioneered a fifth, rather different approach to sequentiality, using *logical relations* to eliminate functions that were not obviously sequential. For example, *por* does not preserve the ternary relation

$$R = \{(d, e, f) \in \{\mathit{true}, \mathit{false}, \perp\}^3 \mid (d = \perp) \vee (e = \perp) \vee (d = e = f)\}$$

whereas all the operations of PCF do preserve that relation. This argument, discovered independently and earlier by Plotkin [AC80], was expanded by Sieber to a semantic definition of which fixed, finite-arity relations were preserved by the operations of PCF. Sieber then used the relations to obtain a model that was fully abstract for PCF up to third-order types. O'Hearn and Riecke [OR95b] extended the model with varying arity Kripke logical relations of Jung and Tiuryn [JT93], where each constituent relation was one of Sieber's relations, and proved that the new model was Milner's fully abstract model. They left open the problem of whether Sieber's original model was fully abstract at all types. The situation is rather like [Plo80], where binary relations characterize λ-definability in the full type hierarchy over an infinite ground set up to type-level two, and Kripke relations characterize definability at all types, but it remains open whether binary relations suffice for definability.

Directions

With more than 25 years of research, we still do not have an adequate description of sequential, determinate computation. Even the most successful descriptions have not been generalized much past PCF. Only the games models have been extended, to the lazy λ-calculus [AM95] and to a call-by-name language with recursive types [AM94], and nothing is known about call-by-value languages or languages with categorical or "smash" sums. This unfortunate fact shows that while the sequentiality problem is robust, our understanding of sequentiality is not.

Even if we focus on describing PCF's sequential computation, there are still large gaps in our understanding. The two good semantic descriptions of PCF, the games and logical relation constructions, do not help us answer three related decidability problems first pointed out by Jung and Stoughton [JS93]:

1. Definability problem: Given an element of the monotone model over the booleans, is it PCF definable?
2. Counting problem: Given a PCF type over only the base type `bool`, how many elements are in the fully abstract model at that type?
3. Equality problem: Given two PCF terms whose types involve only the base type of `bool`, are the two terms operationally equivalent—or, equivalently—are the terms equivalent in the fully abstract model?

One can formulate similar recursive enumerability problems for compact elements in the fully abstract model of PCF over the naturals. For PCF over the booleans with *por*, both (2) and (3) are decidable, since the fully abstract model boils down to the monotone function model over a three point cpo. The domain-theoretic approach continues to hold interest precisely because it may answer these questions for PCF. If, however, the problems turn out to be *undecidable*, it may spell doom for the domain-theoretic approach, since we expect the conditions on finite posets and functions to be decidable.

One final question is in order: how good is the original model used by Scott? After the extraneous top elements of the base types are eliminated (see, e.g., [Blo90] for an account of why such elements ought to be eliminated), the model appears to classify correctly many equations. Indeed, the counterexample above due to Plotkin relies on having divergence built into the terms that are operationally equivalent but denotationally distinct. Since programmers (hopefully) do not write divergent subterms, are there counterexamples to full abstraction where divergence is not necessary? At what level of the type hierarchy do such examples occur? Answers to these questions may tell us where reasoning principles for programs can be derived from simpler principles.

Parametric Polymorphism

Peter O'Hearn

An example of a polymorphic function is the function

$$map : \forall \alpha\beta \,.\, (\alpha \to \beta) \to list[\alpha] \to list[\beta]$$

that takes a function and a list as arguments and returns the list obtained by applying the function to each element in the list. A polymorphic function such as this can work for a variety of types α and β, but it is not so unconstrained as to be typeless.

Polymorphism has posed a severe challenge for semantics from the beginning. Initially, the most substantial issue faced was the *impredicativity* of the polymorphic, or second-order, λ-calculus, an extension of typed λ-calculus discovered independently by Jean-Yves Girard and John Reynolds in the early seventies [Gir72, Rey74]. More recently the focus has shifted to *parametricity*, the idea that a parametric polymorphic function works uniformly for any types to which it is instantiated: the map function is a good example of this.

Models of Second-Order λ-Calculus

In the context of the polymorphic λ-calculus impredicativity refers to the non-stratified nature of the types in the language. Intuitively, a polymorphic type $\forall\alpha\,.\,T(\alpha)$ is semantically a certain form of indexed product $\prod_{D\in Type} T(D)$. But the rules of the polymorphic calculus allow the type variable α to be instantiated to the same polymorphic type, so that if p has type $\forall\alpha\,.\,T(\alpha)$ then the polymorphic application $p[\forall\alpha\,.\,T(\alpha)]$ has type $T(\forall\alpha\,.\,T(\alpha))$. This indicates a certain circularity in the product $\prod_{D\in Type} T(D)$, because it seems that this product must itself be an element of the indexing collection $Type$. So the question of mere existence of (non-syntactically-defined) models is non-trivial.

Domain Models. One successful attack on impredicativity uses the technology originally developed for solving recursive domain equations. The earliest such model [McC79] interprets types as operators on a suitable universal domain. An alternative, that avoids universal domains, is the model construction given by Girard [Gir86] using ideas from stable domain theory; it was later adapted by Coquand, Gunter and Winskel [CGW89] to a more traditional cpo setting. The construction works by interpreting types as continuous functors of a certain kind, allowing a type to be constructed out of finite approximations. This finitary nature of the interpretation is what allows impredicativity to be skirted. Although the construction does use domain theory technology, the model does not rely on a previously-constructed model of untyped λ-calculus. In this sense the model may be said to provide a genuinely typed viewpoint on the calculus.

Realizability Models. In models based on realizability (in the sense of realizability semantics of intuitionistic logic) types denote "sets" of realizers, typically taken from a previously-constructed model of untyped λ-calculus. The "functions" taken are then those determined in a suitable sense by the untyped realizers.

The existence of such models for the polymorphic calculus dates back in fact to Girard and Troelstra in the seventies. But in the mid eighties there was a discovery, due to Eugenio Moggi and Martin Hyland [LM91, Hyl88], of a perspective on these models that exposed their extraordinary character. Often, concretely, the denotation of a type is given as a PER (partial equivalence relation) over a partial combinatory algebra; in PER models any two elements of an equivalence class are viewed as different realizers of an element of the type. There is a "standard" PER model, where types are PERs over the natural numbers with partial combinatory algebra structure given by Kleene's application $app(m, n) = \varphi_n(m)$ of partial recursive functions. But this concrete description can also be understood as the "externalization" of an internal category, a category that lives inside another category.

One remarkable fact about this second form of description is that the standard PER model appears as a set-theoretic model, provided that one understands "set theoretic" liberally enough allow models of intuitionistic set theory; the model of intuitionistic ZF that contains the PER model in this way is the effective topos [Hyl82, Ros90]. From the point of view of intuitionistic set theory the result is models where the function type $A \to B$ is interpreted by all set-theoretic functions and $\forall$ is an indexed product $\prod_{D \in Type} T(D)$. This is particularly startling, because Reynolds [Rey84] had earlier shown the impossibility of a (classical) set-theoretic model, a model where types denote sets of some kind and where the function type $A \to B$ consists of all set theoretic functions, with "all set theoretic functions" understood classically.

The other remarkable thing is that these internal categories are, in a suitable sense, small (set-sized) and complete (closed under all small products and other limits). Completeness is very powerful: it makes reasonably obvious that a variety of type theories can be modelled with ease. It is crucial, however, that "completeness" is understood relative to the ambient category, such as the effective topos or one of its subcategories, and not in the usual classical (external) sense. Important work illuminating these issues include:

- Andy Pitts's [Pit87] demonstration that enough "intuitionistically set-theoretic" models exist to satisfy a completeness theorem (in stark contrast to the classical case); and
- the study by Hyland, Edmund Robinson and Pino Rosolini elucidating subtle completeness properties of internal categories in the effective topos and other categories related to it [HRR90] (also, [Hyl82, Rob89, FRR92b]).

Work on realizability models has flourished. Specific directions include algebraic characterizations of low-order types, work on recursion, and on subtypes; just a few examples are [HRR88, BFSS90, Ama89, FMRS90, BL90, Mit88]. Other pointers to work on realizability models can be found in the section on synthetic domain theory.

Comparison and Evaluation. The construction and understanding of models of the polymorphic λ-calculus represents a substantial achievement. With this understood, it is worthwhile to consider a number of criticisms of the domain and PER/realizability approaches.

The strongest criticism of domain models is that none of them are *parametric*, and it is not clear how they may be modified to be so. We will consider parametricity in more detail below: let us simply mention, for now, that the types contain "junk," elements that are non-uniform and contradict the spirit of parametric polymorphism. For instance, in Girard's model the type $\forall\alpha\,.\,\alpha \times \alpha \to \alpha$ consists of four elements. Three of these – $\perp$ and the two projections – are perfectly reasonable, and definable in the polymorphic calculus with recursion. The fourth element is the function that takes two arguments a, b and returns their meet $a \sqcap b$. The meet exists because the types are coherent spaces, but it is easy, using relational parametricity with complete relations (see below), to explain the sense in which it is not parametric.

A second problem is that the domain models can all interpret the polymorphic λ-calculus *with* a fixed-point operator. Reynolds had argued in [Rey83] that "types are not limited to computation," and that that the polymorphic calculus should possess a set-theoretic model; there seems no prior reason why recursion should be *necessary* in interpreting the calculus. In contrast, realizability models *are* set-theoretic, provided one is willing to accept intuitionistic sets. Note that this does not imply any foundational commitment.

But a conceptual disadvantage to realizability models is (arguably) their reliance on a previously-constructed model of untyped λ-calculus; this should be compared especially with Girard's domain model. So even in the "set-theoretic" models recursion sneaks in the back door. A more practical problem with realizability models is that denotations of terms are usually huge equivalence classes of realizers, and these can be unwieldy to work with.

Of all the work on semantics of polymorphism, one of the more important applications to have emerged is the use of realizability models as tools for experimenting with type theories. One of the fastest ways to show soundness of type rules, or just test out ideas, can be to try to construct a model: realizability models are well-suited to this, primarily because of their strong completeness properties. A prime example of this is in work on applying type-theory technology to develop secure type systems for object-oriented programming (e.g. [BM92]).

Parametricity

When Strachey introduced the notion of polymorphic function in 1967, he immediately distinguished between *parametric* and *ad hoc* polymorphism. Ad hoc functions may work differently at different types, whereas parametric functions are supposed to be uniform. It is this uniformity notion that is increasingly being seen as important to capture semantically. The most satisfactory approach to parametricity thus far is based on logical relations, relations defined by induction on types [Mit90b], and is often referred to as relational parametricity or Reynolds parametricity [Rey83].

Relational Parametricity. Throughout his work Reynolds has emphasized a connection between parametric polymorphism and *representation independence*, the principle that the behaviour of an abstract data type is invariant under changes to its concrete representation. For example, a client that uses a type of stacks should not be able to distinguish (at a suitable level of abstraction) an implementation based on lists from one based on functions with integer domain.

The basic idea behind relational parametricity is simple. Suppose we have a polymorphic function $p : \forall\alpha\,.\,T(\alpha)$. This function can be instantiated to a variety of types, yielding $p[D] : T(D)$, $p[E] : T(E)$... Relational parametricity says that the different instantiations bear the following uniform relationship, which we call the (binary, relational) parametricity condition:

> for any types D and E and any relation $R : D \leftrightarrow E$, there is an induced relation $T(R) : T(D) \leftrightarrow R(E)$, and $(p[D], p[E]) \in T(R)$.

Intuitively, we may regard a relation $R : D \leftrightarrow E$ as relating different representations of α, and $T(R)$ as an invariant relationship that must be maintained. Typically, the relation $T(R)$ is determined in the usual inductive manner of logical relations, with the significant caveat that free type variables other than α are mapped to identity relations. The idea is that two pieces of code satisfying invariant $T(R)$ should behave equivalently from the point of view of the "visible" types, types other than α.

The parametricity condition is stated informally; it is the job of mathematical formulations to make precise terms like "type" and "relation" in this statement. But to get an idea of the constraining effect it has, consider a specific type $\forall\alpha\,.\,\alpha \to \alpha \to \alpha$. If this were interpreted straight as a product $\prod_{D\in Set}\,.\,D \to D \to D$ then it would be far too large to be a set. However, with the relational parametricity condition, taking "types" as sets, we consider only those elements of the product that preserve all relations, where two functions (f, g) are related by $R \to R \to R$ iff for all (a_0, b_0) and (a_1, b_1) in R, $(fa_0a_1, gb_0b_1) \in R$. There are only two such elements of the indexed product, $\lambda xy.x$ and $\lambda xy.y$; to see this use relations $R : 2 \leftrightarrow D$ between a two-point set 2 and any other set D. So the parametricity condition in this case cuts a proper class down to a two-point set. But because of the result of [Rey84], it is not possible to use the parametricity constraint to pick out a suitable subset of $\prod_{D\in Set}\,.\,T(D)$, for *any* type T in the polymorphic λ-calculus: parametricity does not give us a way to skirt impredicativity.

Data Abstraction and Algebra. Parametricity gives a method for proving the equivalence between different representations of an abstract data type. Consider an abstract type α with operations x_i of type $t_i(\alpha)$, $i = 1,..n$. To equate two concrete representations $[T]\, K_1 \ldots K_n$ and $[T']\, K_1' \ldots K_n'$ of the type it suffices to find a relation $R : T \leftrightarrow T'$ under which each K_i, K_i' is invariant (according to the induced relation $t_i(R) : t_i(T) \leftrightarrow t_i(T')$. For instance, we can implement stacks of integers using $list[int]$ as the representation type, or a type $(int \to int) \times int$ using the int component to indicate the top of the stack. The relation used to prove equivalence of the representations relates a list to a pair (f, n) such that $f(0), \ldots f(n)$ is the list.

This reasoning method can be derived from the parametricity condition using either Reynolds' treatment of abstract types using polymorphic application – e.g.

$$(\Lambda\alpha\,.\,\lambda x_1, \ldots, x_n\,.M)\,[T]\,K_1 \ldots K_n$$

to bind α and x_i to their concrete representations – or the treatment later given by Mitchell and Plotkin using existential types (see [Mit86, PA93]).

Put this way, relational parametricity appears as a systematization and generalization of (often informal) ideas for reasoning about data types, objects, and so on (e.g. [Hoa72]), and this connection with data abstraction is part of its appeal. But there is also substantial theoretical support in the way of the many consequences of the (binary, relational) parametricity condition.

- Certain categorical data, such as sums and products, can be encoded in a strong sense using only $\forall$ and $\to$. For example,

  $$\begin{aligned} A + B \;&\equiv\; \forall\alpha\,.\,(A \to \alpha) \to (B \to \alpha) \to \alpha \\ A \times B \;&\equiv\; \forall\alpha\,.\,(A \to B \to \alpha) \to \alpha \end{aligned}$$

  Without parametricity, these formulae only define weak sums and products.

- Existential quantification is coded as

  $$\exists\alpha\,.\,T(\alpha) \;\equiv\; \forall\beta\,.\,(\forall\alpha\,.\,T(\alpha) \to \beta) \to \beta$$

- Any type $T(\alpha)$ with α occurring only positively determines (in a suitable sense) a covariant functor on types, and we get encodings of initial T-algebras and final T-coalgebras:

  $$\begin{aligned} \mu\alpha\,.\,T(\alpha) \;&\equiv\; \forall\alpha\,.\,(T(\alpha) \to \alpha) \to \alpha \\ \nu\alpha\,.\,T(\alpha) \;&\equiv\; \exists\alpha\,.\,\alpha \times (\alpha \to T(\alpha)) \end{aligned}$$

  This determines the structure of a range of types. For instance,

  $$\forall\alpha\,.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$$

  is isomorphic to $\forall\alpha\,.\,((1 + \alpha) \to \alpha) \to \alpha$ using standard manipulations, and this shows that the Church numerals are an initial fixed-point for $T(\alpha) = 1 + \alpha$, as one expects. Again, without parametricity the Church numerals are only weakly initial.

These properties are nowadays usually taken as definitive properties of parametricity, not just relational parametricity, and provide a useful test for any proposed alternative definitions. The importance of these properties can be seen in a number of works, beginning with [Rey83] and continuing in a number of places (e.g. [RP93]), with probably the most systematic exposition being [PA93]. Bainbridge, Freyd, Scedrov and Scott [BFSS90] were the first to define a parametric model satisfying these properties; they achieved this by trimming down the PER model. Of course, for the statements of the properties, and the parametricity condition itself, to make precise sense we need to work either with a specific model or within a more general mathematical framework.

Mathematical Frameworks. In formalizing the notion of relational parametricity it is reasonable to separate the parametricity condition itself from the problem of existence of a model. That is, independently of whether relations have been used to construct a model, one can ask whether all of the elements that live in it satisfy the relational condition. Some approaches to frameworks for parametricity are the following

1. Reynolds and Ma [MR92] give a categorical formulation of parametricity, in the context of indexed-category models of polymorphism [See87]. There is also similar work of Mitchell and Scedrov [MS92] for essentially ML-style polymorphism.

2. Robinson and Rosolini [RR94] give a related development, with the definitions reworked in the setting of internal category models of polymorphism.

3. Wadler [Wad89] does the same in the context of type-frame models [BMM90], and provides many useful examples of reasoning with relational parametricity (see also [Has94]).

4. Plotkin and Abadi [PA93] define a logic that allows quantification over, and substitution of, relations as well as types, leading to a logical expression of the relational parametricity condition.

In the first three cases we have a notion of what counts as a model of polymorphism, and additional requirements that parametric models must satisfy; [MR92, RR94] contain useful discussions on relationships among the three approaches. One might expect that parametric models in the Ma-Reynolds or Robinson-Rosolini senses would provide sound models of the logic of Plotkin-Abadi, but the details of this have not appeared as of yet. The question of completeness of this or a related logic wrt categorical models is also pertinent.

Constructing Particular Models. One way to obtain a parametric model is to take an existing model, or model construction, where a notion of "product" $\prod_{D\in Type} . T(D)$ is already understood, and then use the parametricity condition to cut down, accepting only certain "elements" of such a product. We do not mean the product literally here, but simply the interpretation of a polymorphic type.

This idea is subtler than first appears. Since the interpretation of a polymorphic type is not literally a (set-theoretic) product, there is the question of the existence of the trimmed down "type"

$$\{\, p \in \prod_{D\in Type} T(D) \mid parametric(p) \,\}$$

for a suitable predicate $parametric(\cdot)$ based on the relational parametricity condition. For instance, a simple-minded attempt to try this construction with domain-theoretic models runs into apparent, but not well-understood, difficulties. Domain models typically rely on *algebraicity* of their cpo's, whereas many of the non-parametric elements that are excluded by this scheme are in fact *finite* elements. So it is not at all obvious that algebraicity is preserved.

But there are circumstances where this "cut-down scheme" can work. One case is for predicative type theories. In these theories types are stratified into levels, with quantifiers ranging over types from lower levels. Martin Löf's type theories and the core ML type system are examples. For instance, in ML a polymorphic type cannot appear to the left or right of $\rightarrow$: occurrences of $\forall$ are outside of all other connectives. As a result, it is possible to interpret $\forall\alpha\,.\,T(\alpha)$ literally as a trimmed-town set: because of predicativity, the indexing collection $Type$ can be taken to be small, and there are no foundational difficulties. This can be done with sets or, if one considers recursion, with cpo's by taking $Type$ to be, say, the countably-based bounded-complete algebraic cpo's (which is essentially a small collection) and $parametric(p)$ to be determined by relations between cpo's that are strict and have lubs of directed subsets.

For the full second-order polymorphic calculus the cut-down scheme works for models that have sufficient completeness properties. Completeness here means "enough" limits, which essentially allows one to interpret the comprehension in

$$\{p \in \prod_{D \in Type} T(D) \mid parametric(p)\}$$

while maintaining other properties required of models. The first parametric model was in fact obtained by taking the standard PER model, and applying the cut-down scheme wrt certain relations between PERs [BFSS90]. The key fact is that PERs are sufficiently rich to allow the trimmed-down PER to exist. General conditions sufficient for collapsing to a parametric model, together with a slick model construction based on internal categories, have been given in [RR94].

As an alternative to the cut-down scheme, one might hope for a model that is somehow more inherently parametric, that is to say, a model whose definition does not explicitly use logical relations, but for which all elements satisfy a parametricity condition. Such a semantics would be very attractive but, apart from syntactically-defined models [BTC88, Has91], no such are known for the full polymorphic calculus or for any type theory where polymorphic functions can be passed as arguments. One interesting bit of progress is the game semantics of Samson Abramsky and Radha Jagadeesan [AJ94a], which gives a model of multiplicative linear type theory in which all elements are definable (and one expects parametric). This covers a weak form of polymorphism related to that found in ML, and the model construction does not use logical relations.

Other Approaches to Parametricity. Relational parametricity expresses a uniformity property of a family of functions in terms of relationships between instances of the family. A more direct approach might be to say that parametric functions are those that are given by uniform algorithms, algorithms that "work the same way" at all types. An extreme take on this idea is to posit that "working the same way" is tantamount to being a typeless algorithm that happens to behave in a type-correct fashion for all instantiations of a polymorphic type. This notion is what Plotkin and Abadi term Strachey Parametricity, to contrast with Reynolds Parametricity ([PA93], also [Mit90a]).

The standard PER model of polymorphism furnishes something of a formalization of this notion. A polymorphic type $\forall\alpha.T(\alpha)$ is interpreted as an infinitary intersection $\bigcap_{A \in PER} TA$, so that a realizer for a polymorphic function is, in effect, a typeless function that is type-correct for all instantiations. But this connection between Strachey Parametricity and PER models does not extend to all realizability models of polymorphism, or even all PER models; the interpretation of $\forall$ as intersection is highly dependent on the specific nature of the "internal set of PERs" in the ambient category of realizability.

Another idea that deserves mention is dinaturality. Given categories C and D, and functors $F, G : C^{op} \times C \to D$, a dinatural transformation from F to G is a family of maps $m_A : F[A, A] \to G[A, A]$ satisfying the famous hexagon property:

$$\forall f : A \to B \,.\, G[id_A, f] \circ m_A \circ F[f, id_A] \;=\; G[f, id_B] \circ m_B \circ F[id_B, f]$$

The C^{op} component is used to take care of negative occurrences of type variables; dinaturals are an adaptation of natural transformations to account for mixed-variance. A connection between dinaturality and parametricity was proposed in [BFSS90].

The notion of dinatural has problems: dinatural transformations do not compose in general, and thus one does not automatically get a category. However, all definable elements in the polymorphic λ-calculus are dinatural, so dinaturality can be used as a sound principle for reasoning about polymorphic functions. It was further shown in [PA93] that relational parametricity implies dinaturality, in the context of their logic for polymorphism.

But dinaturality has an important feature (apart from its simple expression) that present treatments of relational parametricity lack: it is formulated generally, for any categories C and D, and any functors $C^{op} \times C \to D$. (This does not take nested polymorphic functions explicitly into account.) Thus, the notion is precise, but not tied in any way to particular type theories. Peter Freyd's work on structors also a general flavour of this kind [Fre93].

Directions and Problems. Broadly speaking, there is room and reason to investigate both generalizations and further applications of parametricity. For example, it has been suggested that the "information hiding" aspect of locally encapsulated state, as found in objects or higher-order imperative programming, is closely related to parametricity [OT95]. It is reasonable to ask whether there is a general notion of which the "abstractness of interfaces" found in polymorphic type theories, in imperative and object-oriented programming, and in many other programming situations are instances.

There remain specific questions about relational parametricity and related approaches. The connection between data abstraction and uniformity has not been completely explained [Rob], and there are particularly questions about alternative formalizations. For example:

- *Is the PER model relationally parametric?* Is the trimmed-down model of [BFSS90] any different than the standard PER model? Answering this would be a first step to relating relational parametricity and Strachey parametricity. (Some partial results are [HRR90, FRR92a].)
- *For what n and m are n-ary and m-ary relational parametricity different?* Similarly for Kripke relations [Plo80, JT93].

Most of the discussion so far has avoided fixed-points. Certainly relational parametricity and a polymorphic fixed-point operator $Y : \forall\alpha\,.\,(\alpha \to \alpha) \to \alpha$ can live together comfortably; one restricts to suitable "admissible" relations. But the encoding of initial algebras given above ceases to work (consider $T(\alpha) = \alpha$), and one loses strong sums, so, while parametricity can still be used, the story being told is not nearly so neat. As a remedy Plotkin has proposed that fixed-points should treated in the context of linear polymorphic type theory, instead of the usual intuitionistic calculus. Then one recovers recovers initial algebras via $\mu\alpha\,.\,T(\alpha) \;\equiv\; \forall\alpha\,.\,(T(\alpha) \multimap \alpha) \to \alpha$, one obtains strong sums (which are incompatible with intuitionistic, or Cartesian closed, type theory and recursion), and most remarkably, one obtains solutions of arbitrary domain equations, and not just covariant ones. (This uses Freyd's reduction of recursive to inductive types [Fre90].)

These results have been presented by Plotkin in lectures [Plo93], though they have not been published yet. But the point that linear type theory allows for a better treatment of parametricity with recursion rings clear, and there are likely to be further applications of linear type theory as far as parametricity is concerned. One example already is in work on applying parametricity to the semantics of imperative languages [OR95c].

One way to find a parametric model of impredicative type theory with recursion would be to trim down a domain-based PER model [Ama89, Pho90, FMRS90]. But since we already know of domain-theoretic models of polymorphism, models that accommodate fixed-points, there is a natural question:

- *Can a cpo model of the polymorphic λ-calculus be modified to be parametric?*.

This explicitly does *not* ask for a domain-theoretic PER model, but one more like Girard's or McCracken's. Related questions can be asked for predicative type theories, but then existence is not itself a problem; rather, there are questions about the resulting structure of the types, such as whether all the types denote are algebraic or effectively given cpo's. Generally, there is still an unanswered question as to whether traditional (especially, effective) domain theory based on cpo's is compatible with parametricity, or whether other flavours of domain theory, as found, e.g., in the sections on synthetic and axiomatic domain theory, are more appropriate.

Mutable State

Ian Stark

Explicit manipulation of state has been with programming languages from the beginning; variables and store simply match the registers and memory of the underlying machine. Similarly, almost every introduction to the denotational approach includes some simple imperative language as a standard example; and when the state consists of a fixed set of global integer variables, it is not hard to describe an accurate model. Typically we have

$$\begin{aligned} \mathit{States} &= \mathit{Locations} \to \mathit{Values} \\ \mathit{Expressions} &= \mathit{States} \to \mathit{Values} \\ \mathit{Commands} &= \mathit{States} \to \mathit{States} \end{aligned}$$

where

$$\mathit{Locations} = \{x, y, \ldots\} \quad \text{and} \quad \mathit{Values} = \{\ldots, -1, 0, 1, 2, \ldots\}.$$

The interpretation of a command might then look like this:

$$[\![x := e]\!]s \;=\; s[x \mapsto [\![e]\!]s] \qquad s \in \mathit{States},$$

where $s[x \mapsto \cdot]$ is the state s with location x updated. The denotation of store as a function from locations to their contents goes back a long way: Strachey attributes it to a suggestion of Burstall in 1964 (see the forward to [Sto77]). Adding loops and procedures makes things more complicated, but the treatment of global state remains much the same.

However, state as a model of the underlying machine is not at all the same as state as a programming tool. Real applications use explicit state in several more sophisticated ways: private local variables to add structure and safety to code; variables storing complex objects such as procedures or function closures; and store that is dynamically created and discarded, to be swept up by a garbage collector. These are all useful features, but their interaction with each other, and other techniques like higher-order functions, can be subtle and surprising. This is reflected in the fact that good mathematical models are hard to construct, and in some cases it may even be difficult to find an operational semantics that is clearly correct.

The problems with denotational semantics come in two degrees. First, it may be hard to find any model at all: as with the storage of functions, which is enough on its own to encode recursion and non-termination. Second, a model may not be very abstract, in that it makes distinctions between too many programs. This shows up because explicit mutable state is often most useful when its details can be concealed. Local variables are convenient precisely because they are invisible outside the procedure that uses them; and a memo function, that caches previous results, should appear externally no different from its non-caching version.

We consider here two distinct approaches to mutable state in language design, and review work on the denotational semantics of each. The first concerns Algol-like languages, that try to mix imperative control and higher-order procedures in a safe and even-handed way; the second looks at Standard ML, a broadly functional language with some powerful imperative features. Although these languages encourage quite different styles of programming, there are many similarities in the problems raised and the mathematical models used to understand them.

Idealized Algol

Reynolds has promoted the investigation of Algol-like languages, with block structured use of local variables, call-by-name parameter passing and higher-order procedures. These all come together in *Idealized Algol* [Rey81]. This language makes a sharp distinction between commands, which can modify the state but not return values, and expressions, which can return values but cannot affect the state. As a consequence, all local variables can be safely allocated on a stack. Such carefully drawn constraints also make it possible to consider construction of a denotational semantics for state in Idealized Algol.

One line of models for Algol-like state has been developed by Halpern, Meyer, Trakhtenbrot and Sieber [HMT84, MS88, Sie93, Sie94]. These build on the very simple model described above, but take into account the *support* of procedures: the locations they actually use and how they affect them. This is done through a system of multiple simultaneous logical relations, indexed in the style of Kripke models. In [MS88] Meyer and Sieber give a range of examples that illustrate the difficulties that can arise when reasoning about local variables, and show how the denotational approach can express *invariants* of the store: properties that a procedure is guaranteed to preserve. Sieber, in [Sie94], gives an impressive proof of full abstraction for the second-order subset of the language. We can gain some idea of the difficulty of reasoning about local state by noting that this proof both subsumes and considerably extends Sieber's previous demonstration that logical relations give a complete account of sequentiality up to third-order in the purely functional language PCF — generally considered a hard problem in its own right [Sie92].

Following Reynolds work with Oles [Rey81, Ole82, Ole85], models for Idealized Algol using *functor categories* have been further developed by Tennent, O'Hearn and Lent [OT92, OT95, Len93]. Functors are important because they capture the fact that the size of the store, as well as its contents, may change over time. Thus an index category of *possible worlds* or *state shapes* is used to record what freedom the state has to vary, at any point during program execution. This approach differs from those above, in that explicit locations are not necessarily involved. Rather, a variable is represented by a pair of values, an expression giving its current content, and an *acceptor* that can change it:

$$\begin{aligned} \mathit{Expressions} &= \mathit{States} \to \mathit{Values} \\ \mathit{Acceptors} &= \mathit{Values} \to (\mathit{States} \to \mathit{States}) \,. \end{aligned}$$

This technique of separating the L-value and R-value of a variable has been much promoted by Reynolds, and allows for features such as conditional variables:

$$(\mathit{if}\ \ \mathit{test}\ \ \mathit{then}\ x\ \ \mathit{else}\ y) \ := \ a+b+c\,.$$

Inspired by relational parametricity for polymorphic functions, O'Hearn and Tennent have adapted this model to use *categories with relations*. These capture the notion that a procedure is polymorphic in those parts of the state to which it has no direct access. The resulting models allow reasoning about state invariants, and can prove all of the tricky examples of Meyer and Sieber [MS88], together with various others. Perhaps most interesting from a mathematical standpoint is that this this rather powerful semantics can be represented quite simply as Oles' original model reformulated in the 2-category of reflexive graphs, rather than the 2-category of sets.

These functor category models have been used to give a denotational account of *specification logic*, Reynolds' extension of Hoare logic to higher-order procedures [Rey82, Ten90, OT93]. Specification logic uses a notion of *non-interference* to avoid the problems of variable aliasing, where procedures may affect each other in unforeseen ways. This property can be hard to check, and with *syntactic control of interference* [Rey78, Rey89] Reynolds proposes a scheme to ensure that such interfering code cannot be written. This too has a denotational semantics within the functor category model [Ten83, O'H93], and indeed a comparison with mathematical models for linear logic has inspired a reworking of the original syntactic scheme [O'H91, OPTT95]. This new version corrects some known difficulties with types, by giving a correct handling of *passive* and *active* procedure parameters. We can see this as a prime example of good denotational semantics feeding back into improvements at the language level.

This link to linear logic has been investigated by Reddy, whose models using coherence spaces bring out the *historicity* implicit in the state: variables are objects, and the history of their local state is threaded through the course of program execution [Red93, Red94, Red96]. Remarkably, an apparently crude bolting together of this model with functor categories, via the Yoneda embedding, gives a good description of both locality and historicity at the same time [OR95a]. The success of this model is not yet entirely understood.

Standard ML

The language Standard ML offers functional programming with imperative features. It is built around a core language that provides strongly typed higher-order functions, possibly polymorphic, with call-by-value parameter passing. Mutable state is incorporated through *references*: dynamically created cells that can be read and written at will. Standard ML is a real language, with a number of implementations and, unusually, a complete published operational semantics [MT91, MTH90].

Programs in Standard ML use mutable state in quite different ways to the Algol-like languages described above. Whereas in Algol, local variables are essential intermediaries in all computation, in ML a purely functional sublanguage is sufficient for most programming. In Algol local variables permeate all code, and much of the reasoning effort is devoted to ensuring that they behave well; giving stack allocation and no covert interference between variables. By contrast, in ML most things can be done without reference cells, and when they are used, it is exactly these interference effects that are most important.

For example, a function may have an associated reference cell that persists from one call to the next, providing an accumulator or a cache. A complex data structure might contain cells that change when it is consulted, to speed up future accesses, as with path compression in the classic union-find algorithm. A group of functions can share references in order to communicate among themselves; or a higher-order function may return a function as result, but keep a link to it through a common reference. Dynamic allocation of cells is also important, to match the dynamic creation of functions by higher-order programs.

Combined with the fact that references may contain functions, structured datatypes, or even further references, it is clear that mutable state in a language like ML provides serious challenges for denotational semantics. On the other hand, such power can have curious and unexpected interactions with features like higher-order functions and exceptions, so it is even more important to find solid reasoning principles to confirm that the examples above do behave as expected.

These applications of references all depend on the *visibility* of cells: who can see them, how they may update them, and whether they can pass this access on to others. For ML references, this issue of visibility transcends the usual notions of scope, and has a complex behaviour of its own. To study it, Pitts has identified the *nu-calculus*, a simply-typed lambda-calculus with dynamically generated *names*. Names are created fresh, they can be compared with each other and passed around, but that is all. Bare as it appears, the nu-calculus captures the essence of visibility, and its operational and denotational semantics have a surprisingly rich structure that mirrors the subtleties of reference behaviour in Standard ML.

Pitts and Stark describe the nu-calculus in [PS93b, PS93a], and outline a categorical semantics for the language. This model uses a *computational monad*, following Moggi's general scheme for extensible denotational semantics [Mog90, Mog91]. The idea is to use a cartesian closed category $\mathcal{C}$ for the functional part of the model, and concentrate all non-standard behaviour in a chosen monad $T : \mathcal{C} \to \mathcal{C}$. Loosely, for every object of values A, there is an object of computations TA; and Moggi's observation is that the requirement that T be a strong monad is enough to capture a wide variety of notions of 'computation'. In addition, the internal language of $\mathcal{C}$ then provides a convenient *computational metalanguage* for equational reasoning.

For the nu-calculus, Pitts and Stark specify some simple additional requirements concerning names and visibility, so that any category satisfying them immediately provides an adequate model for the language. In essence, a generalised element of TA is then a computation 'create some new names and return an element of A, that may depend on them'.

They also give some examples of suitable categories, and two of these are investigated in more detail in [Sta96]. Like the models in the previous section, both use functor categories indexed by possible worlds, where the state is now represented by a finite set of names. The more sophisticated uses categories with relations, from O'Hearn and Tennent's work on relational parametricity; and this model is proved to be fully abstract for expressions of ground and first-order type.

All this is brought together in [Sta94], and extended to a language *Reduced ML* that includes integer references. It turns out that the models of the nu-calculus can be reused to give a denotational semantics for this larger language, simply by defining a new monad from the old:

$$T'A \;=\; (T(A \times S))^S \qquad S = I^N\,.$$

Here S is an object of states, defined from objects I of integers and N of names. The intuition is that a computation can now update the store as well as enlarge it.

This equation seems almost to put us back to the original model of global state; except that now we are working not with sets but in a category $\mathcal{C}$ that smoothly handles dynamic creation of locations. Better models of the nu-calculus give better models for Reduced ML, and these can be further improved by refining the object of states S.

In the work cited, these models of names and state are accompanied by a series of operational methods for reasoning about the nu-calculus and Reduced ML. These are especially interesting in that their development was guided quite clearly by the denotational work: in particular the distinction between computations and values, and the use of logical relations. Thus denotational semantics proves its worth, in giving sound intuition on the operational behaviour of a sophisticated programming construct.

Future Directions

While both approaches above clearly have much in common, there is not yet any satisfactory formal relationship between local variables in Algol and references in Standard ML. Such a connection would assist both areas, and might also cast light on the very different styles of programming each involves: naive translation from one to the other can often take a simple first-order procedure to a tortuous third-order one.

A fully-abstract model of state is an obvious aim, as with any work on denotational semantics. As mentioned earlier, proofs of full abstraction at lower types have been obtained for Idealized Algol and the nu-calculus; one route to extending these might be through logical relations of varying arity, as used by O'Hearn and Riecke to construct a fully-abstract model of PCF [OR95b]. Another method may be the link to linear logic, and even game semantics, suggested by Reddy and O'Hearn.

In the case of ML-style references and the nu-calculus, there is another result related to full abstraction and of independent interest. Purely functional languages like PCF satisfy Milner's *context lemma* [Mil77], which says that the behaviour of an expression in any context $C[-]$ is determined by its behaviour in contexts of the form $[-]V_1 \dots V_n$. The importance of this in a typed language is that all the V_i have types smaller than that of the original expression. In a language with state, the context lemma fails: to observe all the behaviour of a function, it may be necessary to apply it more than once to the same argument, or pass it the result of a previous application. The question then, is to find some level at which we can again express the observable behaviour of expressions of type $(\sigma \to \tau)$ in terms of those of types σ and τ.

Finally, there remains almost unlimited scope for extending this work to further uses of mutable state: local variables that store functions, references that store references, references within structured data, and even the interaction with other features like exceptions and input/output.

We have seen how mutable state has moved from a representation of the underlying machine, first to local variables, a pervasive but regulated language mechanism, and then to references, a powerful but very specific programming tool. Denotational semantics has moved with it, and will surely continue to guide and inform future applications and development.

Acknowledgement

Almost all of this was written while the authors were guests of the Newton Institute for Mathematical Sciences in Cambridge, England. They participated in the programme *Semantics of Computation* organized by Samson Abramsky, Gilles Kahn, John Mitchell, and Andrew Pitts. The authors take this opportunity to express their gratitude to the organizers and the Institute for creating the environment without which a text of this kind could never have been written.

References

[Abr87] S. Abramsky. *Domain Theory and the Logic of Observable Properties.* PhD thesis, University of London, 1987.

[Abr91a] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92:161–218, 1991.

[Abr91b] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[AC80] E. Astesiano and G. Costa. Nondeterminism and fully abstract models. *RAIRO*, 14(4):323–347, 1980.

[ACN90] L. Augustsson, Th. Coquand, and B. Nordstrom. A short description of Another Logical Framework. In *Informal Proceedings of the First Workshop on Logical Frameworks, Antibes*, 1990.

[Adá93] J. Adámek. Data types in algebraically ω-complete categories. To appear in Information and Computation, 1993.

[AJ94a] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543–574, 1994.

[AJ94b] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[AM94] S. Abramsky and G. McCusker. Games for recursive types. In C. Hankin and R. Nagarajan, editors, *Theory and Formal Methods 1994*, Lecture Notes in Computer Science. Springer Verlag, 1994. to appear.

[AM95] S. Abramsky and G. McCusker. Games and full abstraction for the lazy λ-calculus. In *10th Symposium on Logic in Computer Science*, pages 234–243. IEEE Computer Society Press, 1995.

[Ama89] R. M. Amadio. Recursion over realizability structures. *Information and Computation*, 91:55–85, 1989.

[AMJ94] S. Abramsky, P. Malacaria, and R. Jagadeesan. Full abstraction for PCF (extended abstract). In M. Hagiya and J.C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 1–15. Springer Verlag, 1994. (Full version available from http://theory.doc.ic.ac.uk:80/tfm/papers/MalacariaP/PCFfullabs.dvi.Z).

[AO93] S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.

[AP90] M. Abadi and G. D. Plotkin. A Per model of polymorphism and recursive types. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 355–365. IEEE Computer Society Press, 1990.

[Bar92] M. Barr. Algebraically compact functors. *Journal of Pure and Applied Algebra*, 82:211–231, 1992.

[BBHdP93] P.N. Benton, G.M. Bierman, J.M.E. Hyland, and V.C.V. de Paiva. Linear lambda calculus and categorical models revisited. In E. Börger et al., editor, *Selected Papers from Computer Science Logic '92*, volume 702 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[BC85] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: The kernel of the applicative language CDS. In J. C. Reynolds and M. Nivat, editors, *Algebraic Semantics*, pages 35–84. Cambridge University Press, 1985.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: The state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Semantics*, pages 89–132. Cambridge University Press, 1985.

[BE91] A. Bucciarelli and T. Ehrhard. Sequentiality and strong stability. In *6th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1991.

[Ber78] G. Berry. Stable models of typed λ-calculi. In *Proceedings of the 5th International Colloquium on Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, pages 72–89. Springer Verlag, 1978.

[Ber79] G. Berry. Modèles Complèment Adéquats et Stables des Lambda-calculs typés, 1979. Thèse de Doctorat d'Etat, Université Paris VII.

[Ber93] U. Berger. Total sets and objects in domain theory. *Annals of Pure and Applied Logic*, 60:91–117, 1993.

[BFSS90] S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(10):35–64, 1990. *Corrigendum* in 71(3):431, 1990.

[BG93] S. Brookes and S. Geva. Sequential functions on indexed domains and full abstraction for a sub-language of PCF. Technical Report CMU-CS-93-163, Carnegie Mellon University, 1993.

[BJO91] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 91:23–55, 1991.

[BL90] K. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990.

[Blo90] B. Bloom. Can LCF be topped? Flat lattice models of typed λ-calculus. *Information and Computation*, 87:264–301, 1990.

[BM92] K. Bruce and J. C. Mitchell. PER models of subtyping, recursive types, and higher-order polymorphism. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–327. ACM, New York, 1992.

[BMM90] K. Bruce, A.R. Meyer, and J.C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85(1):76–134, 1990.

[Bra94] T. Braüner. A general adequacy result for a linear functional language. Technical Report RS-94-22, BRICS Research Series, August 1994. (Appears in the proceedings of MFPS'94).

[BTC88] V. Breazu-Tannen and T. Coquand. Extensional models of polymorphism. *Theoretical Computer Science*, 59:85–114, 1988.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knobloch, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice Hall, 1986.

[CCF94] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111:297–401, 1994.

[CDCHL84] M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structure and filter lambda models. In G. Lolli, G. Longo, and A. Marcja, editors, *Logic Colloquium '82*, pages 241–262. Elsevier Science Publishers B.V. (North-Holland), 1984.

[CF92] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *Conference Record 19th ACM Symposium on Principles of Programming Languages*, pages 328–342. ACM, New York, 1992.

[CGW89] T. Coquand, C. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81:123–167, 1989.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CP92] R. L. Crole and A. M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.

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

[Cur92] P.-L. Curien. Observable algorithms on concrete data structures. In *Logic in Computer Science*, pages 432–443. IEEE Computer Society Press, 1992.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. R. Hindley and J. P. Seldin, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalisms*, pages 589–606. Academic Press, 1980.

[Eda95a] A. Edalat. Domain theory in learning processes. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *11th Conference on Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*, http://www.elsevier.nl:80/mcs/tcs/pc/Menu.html, 1995. Elsevier Science Publishers B.V.

[Eda95b] A. Edalat. Dynamical systems, measures and fractals via domain theory. *Information and Computation*, 120(1):32–48, 1995.

[Ehr93] T. Ehrhard. Hypercoherences: A strongly stable model of linear logic. *Mathematical Structures in Computer Science*, 3:365–386, 1993.

[Fio93] M.P. Fiore. A coinduction principle for recursive data types based on bisimulation. In *8*$^{\text{th}}$ *LICS Conf.* IEEE, Computer Society Press, 1993. (Full version to appear in Information and Computation special issue for LICS93).

[Fio94a] M. P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. PhD thesis, University of Edinburgh, 1994. To be published by Cambridge University Press in the Distinguished Dissertations Series.

[Fio94b] M.P. Fiore. First steps on the representation of domains (extended abstract). Manuscript available from http://www.dcs.ed.ac.uk (full version in preparation), December 1994.

[FMRS90] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. In *Logic in Computer Science*, pages 346–354. IEEE Computer Society Press, 1990.

[FP94] M. P. Fiore and G. D. Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In *9*$^{\text{th}}$ *LICS Conf.*, pages 92–102. IEEE, 1994.

[Fre90] P.J. Freyd. Recursive types reduced to inductive types. In *5th Symposium on Logic in Computer Science*, pages 498–507. IEEE Computer Society Press, 1990.

[Fre91] P. J. Freyd. Algebraically complete categories. In A. Carboni, M. C. Pedicchio, and G. Rosolini, editors, *Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer Verlag, 1991.

[Fre92] P. J. Freyd. Remarks on algebraically compact categories. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *L.M.S. Lecture Notes*, pages 95–106. Cambridge University Press, 1992.

[Fre93] P. J. Freyd. Structural polymorphism. *Theoretical Computer Science*, 115:107–129, 1993.

[FRR92a] P. J. Freyd, E. P. Robinson, and G. Rosolini. Dinaturality for free. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Proc. LMS Symposium on Applications of Categories in Computer Science, Durham 1991*, volume 177 of *LMS Lecture Note Series*, pages 107–118. Cambridge University Press, 1992.

[FRR92b] P. J. Freyd, E. P. Robinson, and G. Rosolini. Functorial parametricity. In *6th Annual IEEE Symposium on Logic in Computer Science*, pages 444–452. IEEE Computer Society Press, 1992.

[GHK$^+$80] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer Verlag, 1980.

[Gir72] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur, 1972. Thèse d'Etat, Université Paris VII.

[Gir86] J.-Y. Girard. The system F of variable types: Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[GS90] C. Gunter and D. S. Scott. 12: Semantic Domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 633–674. Elsevier Science Publishers, Amsterdam, 1990.

[Gun92] C. Gunter. *Semantics of Programming Languages. Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[Has91] R. Hasegawa. Parametricity of extensionally collapsed models of polymorphism and their categorical properties. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 495–512. Springer Verlag, 1991.

[Has94] R. Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4:71–109, 1994.

[HHP92] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992.

[HJ95] C. Hermida and B. Jacobs. Induction and coinduction via subset types and quotient types. In P. Dybjer and R. Pollack, editors, *Informal proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory*, May 1995. (Available as Technical Report 85, Programming Methodology Group, Göteborg University and Chalmers University of Technology).

[HM95] J. M. E. Hyland and E. Moggi. The S-replete construction. In *Category Theory in Computer Science*, volume 953 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[HMT84] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 245–257. ACM Press, 1984.

[HO] M. Hyland and L. Ong. Dialogue games and innocent strategies: An approach to intensional full abstraction for PCF (preliminary announcement). Cambridge University, UK. Note distributed August 1993.

[HO95a] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for PCF. Manuscript (available from http://theory.doc.ic.ac.uk:80/tfm/papers/OngCLH/pcf.ps.gz), 1995.

[HO95b] J.M.E. Hyland and C.-H.L. Ong. Pi-calculus, dialogue games and PCF. In *7th Annual ACM Conference on Functional Programming Languages and Computer Architecture, La Jolla, California*, 1995.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[HP79] M. C. B. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In J. Beçvar, editor, *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

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

[HRR88] J. M. E. Hyland, E. P. Robinson, and G. Rosolini. The discrete objects in the effective topos. *Proceedings of the London Mathematical Society*, 1988.

[HRR90] J. M. E. Hyland, E. P. Robinson, and G. Rosolini. Algebraic types in PER models. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 333–350. Springer Verlag, 1990.

[Hyl82] J. M. E. Hyland. The effective topos. In A. S. Troelstra and D. van Dalen, editors, *The L. E. J. Brouwer Centenary Symposium*, pages 165–216. North-Holland, 1982.

[Hyl88] J. M. E. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40:135–165, 1988.

[Hyl91] J. M. E. Hyland. First steps in synthetic domain theory. In A. Carboni, C. Pedicchio, and G. Rosolini, editors, *Conference on Category Theory 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 131–156. Springer Verlag, 1991.

[JM91] T. Jim and A. Meyer. Full abstraction and the context lemma. In *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 131–151, 1991.

[Joh77] P. T. Johnstone. *Topos Theory*. Academic Press, 1977.

[Joh82] P. T. Johnstone. *Stone Spaces*, volume 3 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1982.

[Jon90] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, Edinburgh, 1990. Also published as Technical Report No. CST-63-90.

[JP95] A. Jung and H. Puhlmann. Types, logic, and semantics for nested databases. In M. Main and S. Brookes, editors, *11th Conference on Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B.V., 1995.

[JS93] A. Jung and A. Stoughton. Studying the fully abstract model of PCF within its continuous function model. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 230–244. Springer Verlag, 1993.

[JT93] A. Jung and J. Tiuryn. A new characterization of lambda definability. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 245–257. Springer Verlag, 1993.

[Jun90] A. Jung. The classification of continuous domains. In *Logic in Computer Science*, pages 35–40. IEEE Computer Society Press, 1990.

[Keg95] M. Kegelmann. Factorisation systems on domains. Extended Abstract, 1995.

[Lam] F. Lamarche. A large cartesian closed category of domains. To appear in: Information and Computation.

[Law64] F. W. Lawvere. An elementary theory of the category of sets. *Proc. Nat. Acad. Sci.*, 52, 1964.

[Law69] F. W. Lawvere. Diagonal arguments and cartesian closed categories. In *Category Theory, Homology Theory and their Applications II*, volume 92 of *Lecture Notes in Mathematics*. Springer Verlag, 1969.

[Len93] A. F. Lent. The category of functors from state shapes to bottomless CPOs is adequate for block structure. In *Proceedings of the 1993 ACM SIGPLAN Workshop on State in Programming Languages*, number YALEU/DCS/RR-968 in Yale University Department, of Computer Science, Research Report, pages 101–119, 1993.

[LM84] G. Longo and E. Moggi. Cartesian closed categories of enumerations for effective type structure, part I and II. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 235–255. Springer Verlag, 1984. Proceedings of International Symposium at Sophia-Antipolis, France, June 27–29, 1984. Lecture Notes in Computer Science Vol. 173.

[LM91] G. Longo and E. Moggi. Constructive natural deduction and its "ω-set" interpretation. *Mathematical Structures in Computer Science*, 1(2):215–254, 1991.

[Lon95] J. Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1995.

[LS81] D. J. Lehmann and M. B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.

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

[MC88] A. R. Meyer and S. S. Cosmadakis. Semantical paradigms: Notes for an invited lecture. In *3rd Symposium on Logic in Computer Science*, pages 236–253. IEEE Computer Society Press, 1988.

[McC79] N. McCracken. *An investigation of a programming language with a polymorphic type structure*. PhD thesis, Syracuse University, 1979.

[McC84] D. C. McCarty. *Realizability and Recursive Mathematics*. PhD thesis, Oxford University, 1984.

[Mil77] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mit86] J. C. Mitchell. Representation independence and data abstraction. In *Conference Record 13th ACM Symposium on Principles of Programming Languages*, pages 263–276. ACM Press, 1986.

[Mit88] J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[Mit90a] J. C. Mitchell. A type inference approach to reduction properties and semantics of polymorphic expressions (summary). In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 195–212. Addison-Wesley, 1990.

[Mit90b] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 365–458. North Holland, 1990.

[ML83] P. Martin-Löf. Lecture notes on the domain interpretation of type theory. In Programming Methodology Group, editor, *Workshop on the Semantics of Programming Languages*, Göteborg, Sweden, 1983. Chalmers University of Technology.

[Mog90] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1990.

[Mog91] E. Moggi. Notions of computations and monads. *Information and Computation*, 93(1):55–92, 1991.

[Mog95] E. Moggi. Metalanguages and applications. Manuscript (available from http://theory.doc.ic.ac.uk:80/tfm/papers/MoggiE/ML-notes.dvi.gz), 1995.

[Mor68] J. H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[MR92] Q. Ma and J. C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer Verlag, 1992.

[MS88] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 191–203. ACM Press, 1988.

[MS92] J.C. Mitchell and A. Scedrov. Notes on sconing and relators. In E. Boerger et al., editors, *Computer Science Logic: 6th Workshop*, volume 702 of *Lecture Notes in Computer Science*, pages 352–378, San Miniato, Italy, 1992. Springer Verlag.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Mul81] P. Mulry. Generalized Banach-Mazur functionals in the topos of recursive sets. *Journal of Pure and Applied Algebra*, 26:71–83, 1981.

[Mul87] K. Mulmuley. *Full Abstraction and Semantic Equivalence*. The MIT Press, 1987.

[Nic94] H. Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at St. Petersburg*, Lecture Notes in Computer Science. Springer Verlag, 1994.

[O'H91] P. W. O'Hearn. Linear logic and interference control (preliminary report). In *Category Theory and Computer Science 1991*, number 530 in Lecture Notes in Computer Science, pages 74–93. Springer Verlag, 1991.

[O'H93] P. W. O'Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3:435–465, 1993.

[Ole82] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages.* PhD thesis, Syracuse University, 1982.

[Ole85] F. J. Oles. Type categories, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Semantics*, pages 543–574. Cambridge University Press, 1985.

[OPTT95] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In M. Main and S. Brookes, editors, *Mathematical Foundations of Programming Semantics: Proceedings of the 11th International Conference*, number 1 in Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B.V., 1995.

[OR95a] P. W. O'Hearn and U. S. Reddy. Objects, interference and the Yoneda embedding. In M. Main and S. Brookes, editors, *Mathematical Foundations of Programming Semantics: Proceedings of the 11th International Conference*, number 1 in Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B.V., 1995.

[OR95b] P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF. *Information and Computation*, 120(1):107–116, 1995.

[OR95c] P.W. O'Hearn and J.C. Reynolds. From Algol to polymorphic linear lambdas calculus. Lectures at Isaac Newton Institute for Mathematical Sciences, Cambridge, 1995.

[OT92] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science 1991*, number 177 in London Mathematical Society Lecture Note Series, pages 217–238. Cambridge University Press, 1992.

[OT93] P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, part 2. *Information and Computation*, 107(1):25–57, 1993.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42:658–709, 1995.

[PA93] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Bezem J. F. Groote, editor, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer Verlag, 1993.

[Pho90] W. Phoa. Effective domains and intrinsic structure. In *5th Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1990.

[Pho91] W. Phoa. *Domain Theory in Realizability Toposes.* PhD thesis, University of Cambridge, 1991.

[Pho94] W. Phoa. From term models to domains. *Information and Computation*, 109:211–255, 1994.

[Pit87] A. M. Pitts. Polymorphism is set theoretic, constructively. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science, Proc. Edinburgh 1987*, volume 283 of *Lecture Notes in Computer Science*, pages 12–39. Springer Verlag, 1987.

[Pit93] A. M. Pitts. Relational properties of recursively defined domains. In *8th Annual Symposium on Logic in Computer Science*, pages 86–97. IEEE Computer Society Press, Washington, 1993.

[Pit94] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[Pit96] A. M. Pitts. Relational properties of domains. *Information and Computation*, 1996. To appear.

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

[Plo80] G. D. Plotkin. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, London, 1980.

[Plo81] G. D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the "Pisa Notes"). Dept. of Computer Science, Univ. of Edinburgh, 1981.

[Plo85] G. D. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford 1985, 1985.

[Plo93] G. D. Plotkin. Type theory and recursion. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, page 374. IEEE Computer Society Press, 1993.

[Plo95] G.D. Plotkin. Algebraic completeness and compactness in an enriched setting. Invited lecture given at LDPL95 (Workshop on Logic, Domains, and Programming Languages), 1995.

[Pol95] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1995.

[PS93a] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer Verlag, 1993.

[PS93b] A. M. Pitts and I. D. B. Stark. On the observable properties of higher order functions that dynamically create local names (preliminary report). In *Workshop on State in Programming Languages, Copenhagen, 1993*, pages 31–45. ACM SIGPLAN, 1993. Yale Univ. Dept. Computer Science Technical Report YALEU/DCS/RR-968.

[Puh95] H. Puhlmann. Re-grouping information in a domain theoretic data model. Preprint, Technische Hochschule Darmstadt, October 1995. 27pp.

[Red93] U. S. Reddy. A linear logic model of state. Technical Report FP-1993-3, Department of Computer Science, University of Glasgow, January 1993.

[Red94] U. S. Reddy. Passivity and independence. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science*, pages 342–352. IEEE Computer Society Press, 1994.

[Red96] U. S. Reddy. Global state considered unnecessary: Object-based semantics for interference-free imperative programs. *Lisp and Symbolic Computation*, 9(1), 1996.

[Reu95] B. Reus. *Program Verification in Synthetic Domain Theory.* PhD thesis, Universität München, November 1995.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer Verlag, 1974.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, 1978.

[Rey81] J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Rey82] J. C. Reynolds. Idealized Algol and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing*, pages 513–523. Elsevier Science Publishers, 1983.

[Rey84] J. C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer Verlag, 1984.

[Rey89] J. C. Reynolds. Syntactic control of interference, part II. In *Proceedings of ICALP '89*, number 372 in Lecture Notes in Computer Science, pages 704–722. Springer Verlag, 1989.

[Rob] E. P. Robinson. Parametricity as isomorphism. *Theoretical Computer Science.* Accepted for publication.

[Rob88] E. Robinson. Logical aspects of denotational semantics. In *Summer Conference on Category Theory and Computer Science, Edinburgh, 7th–9th September 1987*, Lecture Notes in Computer Science. Springer Verlag, 1988.

[Rob89] E. P. Robinson. How complete is PER? In *4th Annual Symposium on Logic in Computer Science*, pages 106–111. IEEE Computer Society Press, 1989.

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

[Ros90] G. Rosolini. About modest sets. *Int. J. Found. Comp. Sci.*, 1:341–353, 1990.

[Ros92] G. Rosolini. An exper model for Quest. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 436–445. Springer Verlag, 1992.

[Ros95] G. Rosolini. Notes on synthetic domain theory. Available by ftp from `theory.doc.ic.ac.uk`, 1995.

[RP93] J. C. Reynolds and G. D. Plotkin. On functors expressible in the ploymorphic typed lambda calculus. *Information and Computation*, 105:1–29, 1993.

[RR88] E. Robinson and G. Rosolini. Categories of partial maps. *Information and Computation*, 79:95–130, 1988.

[RR94] E. P. Robinson and G. Rosolini. Reflexive graphs and parametric polymorphism. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1994.

[RS94] B. Reus and Th. Streicher. Naïve synthetic domain theory. Available via ftp from `ftp.informatik.uni-muenchen.de` in directory `pub/local/pst/papers/streicher+reus`, 1994.

[Sco69] D. S. Scott. A type theoretic alternative to ISWIM, CUCH, OWHY. Manuscript, University of Oxford, 1969.

[Sco70] D. S. Scott. Outline of a mathematical theory of computation. In *4th Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.

[Sco72] D. S. Scott. Continuous lattices. In E. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136. Springer Verlag, 1972.

[Sco76] D. S. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.

[Sco82] D. S. Scott. Domains for denotational semantics. In M. Nielson and E. M. Schmidt, editors, *International Colloquium on Automata, Languages and Programs*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer Verlag, 1982.

[Sco93] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Reprint of a manuscript written in 1969.

[See87] R. A. G. Seely. Categorical semantics of higher-order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52:969–989, 1987.

[Sie92] K. Sieber. Reasoning about sequential functions via logical relations. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Proc. LMS Symposium on Applications of Categories in Computer Science, Durham 1991*, volume 177 of *LMS Lecture Note Series*, pages 258–269. Cambridge University Press, 1992.

[Sie93] K. Sieber. New steps towards full abstraction for local variables. In *Proceedings of the 1993 ACM SIGPLAN Workshop on State in Programming Languages*, number YALEU/DCS/RR-968 in Yale University, Department of Computer Science, Research Report, pages 88–100, 1993.

[Sie94] K. Sieber. Full abstraction for the second order subset of an Algol-like language (preliminary report). Technical Report A 01/94, Universität des Saarlandes, Saarbrücken, January 1994.

[Sim92] A. K. Simpson. Recursive types in Kleisli categories. Manuscript (available from http://www.dcs.ed.ac.uk/), 1992.

[Smy77] M. B. Smyth. Effectively given domains. *Theoretical Computer Science*, 5:257–274, 1977.

[Smy83a] M. B. Smyth. The largest cartesian closed category of domains. *Theoretical Computer Science*, 27:109–119, 1983.

[Smy83b] M. B. Smyth. Powerdomains and predicate transformers: a topological view. In J. Diaz, editor, *Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 662–675. Springer Verlag, 1983.

[Smy91] M.B. Smyth. I-categories and duality. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 270–287. Cambridge University Press, 1991.

[SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing*, 11:761–783, 1982.

[Sta94] I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994. Also published as Technical Report 363, University of Cambridge, Computer Laboratory.

[Sta96] I. Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1), 1996.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman/Wiley, 1988.

[Sto90] A. Stoughton. Equationally fully abstract models of PCF. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 271–283. Springer Verlag, 1990.

[Sto91] A. Stoughton. Interdefinability of parallel operations in PCF. *Theoretical Computer Science*, 79:357–358, 1991.

[Tay91] P. Taylor. The fixed point property in synthetic domain theory. In *6th LICS conference*, pages 152–160. IEEE Computer Society Press, 1991.

[Tay95] P. Taylor. An abstract stone duality. Available by ftp from `theory.doc.ic.ac.uk`, 1995.

[Ten83] R. D. Tennent. Semantics of interference control. *Theoretical Computer Science*, 27:297–310, 1983.

[Ten90] R. D. Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, 1990.

[TP90] P. Taylor and W. K.-S. Phoa. The synthetic plotkin powerdomain. Available by ftp from `theory.doc.ic.ac.uk`, 1990.

[Vic89] S. J. Vickers. *Topology Via Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Wad89] P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, New York, 1989.

[Wan79] M. Wand. Fixed point constructions in order-enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. MIT Press, 1993.

[Win94] G. Winskel. Stable bistructure models of PCF. Technical Report RS-94-13, BRICS, 1994.

[Zha91] G.-Q. Zhang. *Logic of Domains*. Progress in Theoretical Computer Science. Birkhäuser, 1991.