Domain Theory meets Interaction

Glynn Winskel

Huawei Research Centre, Edinburgh, UK

Abstract

Domain theory and denotational semantics were invented in the late 1960's by Christopher Strachey and Dana Scott. They provided the first mathematical foundation for the semantics of programming languages and their analysis tools. Domain theory has been extremely influential and is still an active area of research especially in its extensions to probabilistic, quantum programming and differentiable programming. This article traces domain theory from its origins as a theory of computable functions, through its limitations, to recent connections with interactive computation via concurrent games and strategies.

1 Introduction

After more than 50 years of development, domain theory permeates computer science. While it has its limitations it has set a compelling paradigm in the formalisation and analysis of computation. It interprets types as domains and programs as (continuous) functions. In many contexts its models and reasoning techniques are the simplest we could hope for. If alone for this reason, it is here to stay. It grew up accompanied by the methodology of denotational semantics. This methodology, of giving meaning to programming languages and systems in a compositional fashion, is the only way to manage their complexity; and its achievement tests our understanding and the robustness of our models.

A word on the historical importance of domain theory, which I think is sometimes forgotten in the rush of practice and innovation. Domain theory and denotational semantics have played key roles in many areas, for instance: they were critical in the evolution of functional programming languages; continuations (widely used in compilation) originated there; their cousin, abstract interpretation, sharing the same techniques, is at the foundation of many static analysers; linear logic, through domain theory, provides the important theory behind the popular system-programming language Rust. As I hope will become clear, the techniques of domain theory and denotational semantics marry well with interactive computation and are still highly relevant today.

Domain theory grew out of a functional way of understanding computation. It is no surprise that it began to meet its limits with the shift to a more interactive form of computation. While the functional paradigm has a long history, theories of interactive computation have been more unsettled. The article concludes with an indication of recent work which combines the methodology of domain theory with interaction through distributed/concurrent games; the role of domains is replaced by games and that of functions by strategies, so types are games and programs are strategies.

Central to any compositional theory of interaction is the dichotomy between a system and its environment. Concurrent games and strategies address the dichotomy in fine detail, very locally, in a distributed fashion, through distinctions between Player moves (events of the system) and Opponent moves (those of the environment). A functional approach has to handle the dichotomy more ingeniously, through its blunter distinction between input and output. This has led to a variety of functional approaches, specialised to particular interactive demands. They include: Girard's Geometry of Interaction; Gödel's Dialectica interpretation; lenses and optics; and the latter's extensions to containers in dependent lenses and optics. Surprisingly, at least for the author, domain-theoretic expressions of these functional paradigms arise from special, rather simple, cases of concurrent games.

Should the reader wish to follow up by reading more on domain theory and its history there is *e.g.* the author's textbook "The formal semantics of programming languages" which is also available in Chinese [1]. For a more advanced treatment of domain theory see the handbook chapter [2]. The later part of this article, on concurrent games and strategies and their relation with functional paradigms, is expanded on informally in a newsletter [3] and more technically in [4].

I have tried to be as informal as possible in writing this article. Where I've added technical or additional parts for more precision they are indicated by being in italic font, so they can be skipped more easily.

2 Early beginnings

The history of domain theory is fairly well known. In the mid 1960's Christopher Strachey realised he needed new techniques to understand the sophisticated programming languages he was developing; that he needed a mathematical model with which to give semantics of programming languages. How else was he to be convinced of the correctness of his programs?

Over lunch, in Wolfson College, Oxford, the physicist Roger Penrose suggested he investigate the lambda calculus, a tool of logicians for describing and reasoning about computable functions. Meanwhile the logician Dana Scott, at Princeton, was highly critical of the untyped nature of the lambda calculus, that it allowed such paradoxical phenomena as self-application, the application of a function to itself, something forbidden in traditional set theory. The paradoxical combinator which allows recursive definitions in the lambda calculus relies on self-application. When Strachey and Scott met there were going to be fireworks, of one kind or another. They got on very well, marking the beginning of their famous collaboration.

Scott persuaded Strachey to move to the safe typed lambda calculus and formalised a Logic of Computable Functions, LCF, as a foundation for Strachey's ambitions. As a logician, Scott was concerned with mathematical foundations right from the start. Scott introduced the idea that types denoted domains, simple forms of topological spaces, built on an order of approximation. Although a computable function can act on an infinite input, for instance a computable function can act on a function on the natural numbers, it can only do so via finite approximations to the input. Accordingly Scott promoted the idea that a computable function be understood as a continuous function from the domain of its input to the domain of its output. On domains the usual topological definition of continuity amounted to the function preserving the approximation order and least upper bounds of chains.

The simplest form of domain is a complete partial order, that is, a partial order $(D, \sqsubseteq_D)$ *of approximation with a least element* $\bot_D$ *and least upper bounds* $\bigsqcup_n d_n$*, a form of limit point, of chains of* $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots \sqsubseteq_D d_n \sqsubseteq_D \cdots$ *in* D*. A function* F *from a domain* $(D, \sqsubseteq_D)$ *to a domain* $(E, \sqsubseteq_E)$ *is continuous if* $F(d) \sqsubseteq_E f(d')$ *when* $d \sqsubseteq_D d'$*, and* $\bigsqcup_n F(d_n) = F(\bigsqcup_n d_n)$ *for any chain* $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots \sqsubseteq_D d_n \sqsubseteq_D \cdots$ *in* D*. Based on earlier ideas of Kleene, a continuous function* F *from a domain to itself has a least fixed point* $\mathrm{fix(F)}$ *constructed as the least upper bound,* $\bigsqcup_n F^n(\bot) =: \mathrm{fix(F)}$*.*

One remarkable feature of domains and continuous functions is that, unlike topological spaces in general, under the pointwise, or Scott order on functions the set of continuous functions from a domain D to a domain E itself formed a domain, the function space $[D \to E]$. More obviously the product of domains $D \times E$, consisting of pairs of elements, formed a domain when ordered coordinatewise. Then, in particular, a recursively defined function from D to E could be readily understood as a least fixed point of the continuous function F on $[D \to E]$ associated with the body of its recursive definition. LCF included a useful inequational logic for reasoning about recursive programs with tools such as Scott induction for establishing properties of least fixed points.

Scott's 1969 article on LCF was circulated widely and very influential. But it was only published relatively much later in 1993 [5]. The reason: Scott had suddenly seen that the techniques for understanding recursive functions as least fixed points could be pushed to the level of types, thus providing a nontrivial domain D isomorphic to its domain of continuous functions $[D \to D]$, so a model of the (untyped) lambda calculus. His objections to the lambda calculus had vanished [6, 7]. Scott's discovery led to a flurry of activity.

3 The word spreads

Researchers outside Oxford joined in the effort. David Park, at Warwick, showed that in Scott's model the paradoxical combinator of the lambda calculus denoted its least fixed point operator. A young Gordon Plotkin, at Edinburgh, and Scott independently discovered universal domains, within which all manner of types could be defined recursively by describing them as certain functions on the domain. Plotkin and Mike Smyth, then a postdoc at Warwick, extended Scott's ideas to a categorical treatment of recursively defined domains. This provided an understanding of a very broad range of recursive types. The mathematical foundations of functional programming were set.

Meanwhile, at Stanford, Robin Milner, Malcolm Newey and Richard Weyrauch had forged ahead with the mechanisation of proofs in LCF [8, 9]. In the process Milner invented the functional language ML as a MetaLanguage to support assisted proofs securely. ML was inspired both by the functional nature of LCF itself and Peter Landin's language ISWIM [10].[1] With its watertight type discipline, ML ensured that only programs yielding legitimate LCF proofs would receive the type "Theorem." At Stanford, Milner began the push into the semantics of concurrent computation through "oracles" to settle nondeterministic choices. These roots continued at Stanford with the work of Zohar Manna and his student Jean Vuillemin on reasoning about recursively defined programs [11].

Domain theory forged new links between computer science and logic. Programming languages and computing systems were amenable to mathematical analysis. Computer scientists approached programming languages with a new confidence born out of a belief that sensible language constructs could be given a mathematical definition. The guidelines of domain theory influenced the design of programming languages and provided a foundation for functional programming.

By the mid 1970's it seemed only a matter of time before domain theory could tackle all features of programming languages. Through continuations Strachey and Christopher Wadsworth had shown how to provide semantics to jumps in imperative languages [12]; building on earlier ideas of Egli and Milner, Plotkin had extended domain theory to a treatment of nondeterministic and parallel programs through his powerdomain [13]—his treatment avoided the non-associativity and non-commutativity of parallel composition Milner had met through using oracles; Nasser Saheb-Djahromi began constructing a probabilistic powerdomain, so enabling the denotational semantics of probabilistic programs. Young researchers in France were beginning to lend their own brilliant vision and mathematical expertise.

4 Limitations—early signs

Domain theory has given us a lasting vision of a mathematical approach to the semantics of programming languages. It supported the method of *denota-*

[1]ISWIM is Peter Landin's joke acronym for "If you See What I Mean."

tional semantics whereby the semantics of a programming language is defined compositionally by structural induction on its syntax.

However, intimations of the limits of domain theory were present in very early work. Gilles Kahn showed how dataflow fitted easily within the scheme; it was a simple matter to represent dataflow processes as continuous functions from streams of input to streams of output, and handle loops in the network through the fixed point treatment of recursion that domain theory provided [14]. But, the extension to nondeterministic dataflow was to be problematic. The difficulties in giving a compositional semantics to nondeterministic dataflow was stressed much later in 1981 by Brock and Ackerman [15, 16]. There are ways to give denotational semantics to nondeterministic dataflow but they lie outside traditional domain theory.

From LCF, Plotkin got the idea of studying its core programming language PCF (Programming Computable Functions), a tradition that continues in testing new theories with some extra feature or other in the presence of function spaces [17]. He invented the concept of *full abstraction*—the name is due to Milner—a way to formalise full agreement between the denotational and operational semantics of a language. Plotkin did this through the vehicle of PCF. Traditional domain theory did not provide a fully abstract model because Scott's domains contained "parasitic" elements such as "parallel or," undefinable in PCF, on which operationally equivalent terms disagreed.

This sparked off the search for more operationally tuned domain theory, one which could capture the sequential evaluation of PCF and lambda-calculi. Both Milner and Vuillemin gave early, slightly different, definitions of what it meant for a continuous function $f : D_1 \times \cdots \times D_m \to E_1 \times \cdots \times E_n$ between products of domains to be *sequential*: for a particular input $(x_1, \cdots, x_m)$ where $f(x_1, \cdots, x_m) = (y_1, \cdots, y_n)$ any increase in an output place y_j had to depend uniformly on an increase in a critical input place x_i. A problem with such a definition is that it depends on the particular way one decomposes a domain into a product.

This was remedied in 1975 by Kahn and Plotkin's definition of *sequential function* between *concrete domains* in which there is an inbuilt notion of place (there called a cell) [18]. Roughly, the elements of a concrete domain consist of a set of events where an event is the filling of a cell with a particular value; as events occur further cells become accessible and able to be filled by at most one of several values. According to Kahn and Plotkin's definition a sequential function between concrete domains is a continuous function for which at any input the filling of an accessible output cell depends on the filling of a critical accessible input cell; there can be several critical cells. The problem was that space of sequential functions wasn't itself a concrete domain. Concrete domains didn't appear suitable for giving a denotational semantics to PCF.

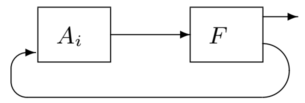
Gérard Berry suggested two remedies. In his first, he proposed restricting continuous functions between domains to those which were *stable*—an approximation to being sequential [19]. Technically, stable functions preserve meets of compatible subsets of elements, but, more intuitively, in a stable function any part of the output depends on a minimum part of the input (reading "part of" as below in the order). Significantly, once the domains are axiomatised appropriately, as what Berry called *dI-domains*, they have function spaces: the set of all stable functions between dI-domains when ordered by a refinement of Scott's order, the *stable order*, themselves form a dI-domain. Roughly, two functions are in the stable order if they are in the Scott order and they share the same minimum inputs when producing common output. Berry went on to consider *bidomains* which possessed both a Scott and stable order. Berry's stable domain theory received an extra impetus in the mid 1980's with Jean-Yves Girard's use first of *qualitative domains* in models of polymorphism [20] and then with his path-breaking discovery of linear logic through special kinds of dI-domains called *coherence spaces* related by stable functions [21].

Stable functions have their own importance, but as a treatment of sequentiality they are just an approximation. With student Pierre-Louis Curien, Berry showed that there was a way to construct a domain theory for sequentiality, within which one could give a denotational semantics to PCF. Though it was at the cost of departing from functions. They showed that concrete domains did indeed have a form of function space if instead of sequential functions they used *sequential algorithms* [22]. Sequential algorithms can be expressed in terms of local decisions as to whether to output a value or inspect a cell for its value. Berry and Curien's pioneering work is a precursor to game semantics, and a sequential algorithm a form of strategy and the decisions its moves, as was laid bare by François Lamarche [23]. However, as Berry and Curien showed, a sequential algorithm can also be viewed as a sequential function together with a function which, given input and a cell accessible at the output, returns a specific critical cell accessible at the input. (This characterisation anticipated lenses in functional programming.)

In sequential algorithms we begin to see a more interactive view of computation, not simply as a the calculation of a function from input to output, but one in which the algorithm actively queries and makes demands on the input, and assigns values to cells. Without it being so obvious at the time, both sequential algorithms and stable functions were part of a growing chorus suggesting computation be based on interaction.

The Brock-Ackerman anomaly of nondeterministic dataflow: While, as Kahn was early to show, deterministic dataflow is a shining application of simple domain theory, nondeterministic dataflow is beyond its scope and, it turns out, an early indicator of the need for a more interactive view of computation. The compositional semantics of nondeterministic dataflow needs a form of generalised relation which specifies the ways *input-output pairs are realised. A compelling example comes from the early work of Brock and Ackerman who were the first to emphasize the difficulties in giving a compositional semantics to nondeterministic dataflow, though our example is based on simplifications in the later work of Rabinovich and Trakhtenbrot [24], and Russell [25].*

Consider the dataflow context below:

$C[A_i]$ =


There are two simple nondeterministic processes A_1 and A_2, which have the same input-output relation, and yet behave differently in the common feedback context $C[-]$, illustrated above. The context consists of a fork process F (a process that copies every input to two outputs), through which the output of the automata A_i is fed back to the input channel, as shown in the figure. Process A_1 has a choice between two behaviours: either it outputs a token and stops, or *it outputs a token, waits for a token on input and then outputs another token. Process A_2 has a similar nondeterministic behaviour: Either it outputs a token and stops,* or *it waits for an input token, then outputs two tokens. For both automata, the input-output relation relates empty input to the eventual output of one token, and non-empty input to one or two output tokens. But $C[A_1]$ can output two tokens, whereas $C[A_2]$ can only output a single token. Notice that A_1 has two ways to realise the output of a single token from empty input, while A_2 only has one. It is this extra way, not caught in a simple input-output relation, that gives A_1 the richer behaviour in the feedback context.*

Over the years there have been many solutions to giving a compositional semantics to nondeterministic dataflow. But they all hinge on a form of generalised relation, to distinguish the different ways in which output is produced from input. A compositional semantics can be given using stable spans*, an extension of Berry's stable functions to include nondeterminism [26]. Stable spans have re-emerged as a special form of concurrent strategy—see Section 8.*

5 Interaction

Concurrent processes can proceed independently but with points of interaction. Their treatment has long been a bugbear of traditional domain theory. While special cases such as deterministic dataflow were easily expressible within domains, and Plotkin's powerdomains with a recursively defined domain of *resumptions* supported parallel composition through the nondeterministic interleaving of actions [13], in general the denotational semantics of parallel programs could seem convoluted. Indeed, after initial excursions into domain models, these complications led Robin Milner to forsake domain theory and denotational semantics in favour of a Calculus of Communicating Systems (CCS) based on Plotkin's structural operational semantics and the process equivalence of bisimulation [27]. Instead, Tony Hoare, with Steve Brookes and Bill Roscoe, proposed a purpose-built domain of failure sets for Communicating Sequential Processes (CSP) [28]. Both Hoare and Milner settled on synchronisation, possibly with the exchange of values, as their primitive of communication. For a number of years concurrency became a rather separate field of study and is still often rather syntax-driven.

Meanwhile since the early 1960's Carl Adam Petri and others had been developing a radically new model of computation, *Petri nets*. Petri nets are based on events making local changes to conditions representing local states. A state of a Petri net is captured by a *marking* which picks out those conditions which currently hold. The net's dynamics, how one marking changes to another, is based on the key idea that the occurrence of an event ends the holding of its preconditions (those conditions with arrows leading to the event) and begins the holding of its postconditions (those conditions with arrows from the event).

The structures of Petri were remarkably similar to those Kahn and Plotkin had uncovered as representations of concrete domains in their investigations of sequentiality. In fact, through the intermediary concept of *event structure*, comprising a set of event occurrences with relations of causal dependency and conflict, Mogens Nielsen, Gordon Plotkin and the author were able to transfer concepts across the two communities, around Petri nets and domains; just as transition systems unfold to trees, so Petri nets unfold to event structures [29, 30]. Notably, Petri's notion of confusion freeness in Petri nets coincided with the restrictions Kahn and Plotkin were making to localise nondeterministic choice to cells. A little later it was realised that Berry's dI-domains were exactly the domains of configurations of event structures ordered by inclusion [31, 32].

In its simplest form, an event structure *is*

$$(E, \leq, \#)\,,$$

comprising a partial order of causal dependency $\leq$ *and a symmetric, irreflexive binary relation of* conflict $\#$ *on* events E*. The relation* $e' \leq e$ *expresses that event* e *causally depends on the previous occurrence of event* e'*. That* $e\#e'$ *means that the occurrence of one event,* e *or* e'*, excludes the occurrence of the other. Together the relations satisfy two axioms: the first axiom says that an event causally depends on only a finite number of events while the second says that events which causally depend on conflicting events are themselves in conflict. A state or history of an event structure is caught in the definition of configuration: a* configurations *of an event structure consists of a subset of events which is down-closed w.r.t.* $\leq$ *and conflict-free. Two events* e, e' *are considered to be causally independent, called* concurrent*, if they are not in conflict and neither one causally depends on the other.*

In diagrams, events are depicted as squares, immediate causal dependencies by arrows and immediate conflicts by wiggly lines. For example,

$$\begin{tikzcd} \square & \square & \\ \square \arrow[u] & \square \arrow[ul] \arrow[u] \arrow[r, squiggly, no head] & \square \end{tikzcd}$$

represents an event structure with five events. The event to the far-right is in immediate conflict with one event—as shown, but in conflict with all events but that on the lower far-left, with which it is concurrent.

But there was a curious mismatch. Whereas Petri nets were largely used to model concurrent processes the corresponding structures in domain theory were being used as representations of domains, so types of processes. The reconciliation of these two views came much later in generalisations of domain theory in which both types and processes denoted event structures—as is the case in concurrent games and strategies.

From the burgeoning world of richly structured models and their equivalences in concurrency, it became clear that concurrent computation wasn't going to fit neatly within traditional domain theory. For instance, a Petri net carried much more structure than could be supported by a point in a domain of information.

Fortunately category theory helped organise models for concurrency: an individual model, say Petri nets, event structures or a transition systems, carried its own style of map to form a category; the maps represented a form of event-respecting simulation and could be used, for instance, to relate the behaviour of a parallel composition to that of its components. Relations between the different categories of models could be expressed as adjunctions; and helped systematise the equivalences on concurrent processes [33, 34]. This separated models of concurrency from the syntax and operational semantics in which they were so often embedded.

For example, a map of event structures from E to E' is a partial function f on events which respects configurations and events: it sends a configuration x of E, by direct image, to a configuration $f\,x$ of E' such that no two distinct events in x go to the same event in $f\,x$. While causal dependency need not be preserved by f, it is reflected locally: if the $e, e' \in x$ and $f(e) \leq f(e')$ then $e \leq e'$. Consequently maps of event structures automatically preserve the concurrency relation on events.

This taxonomy was based on existing models, but it suggested a more general class of models with the versatility to be adapted in the same way as domain theory—a form of generalised domain theory [35, 36]. In several early domain models of processes, a process had been identified with the set of computation paths it could perform. One well-known model of this kind is Hoare's "trace" model of CSP in which a process denotes the set of sequences of visible actions it can perform. The generalised domain theory was similar, but instead of a process being a *set* of computation paths it took a process to be a *presheaf* of computation paths. Roughly a presheaf is like a generalised characteristic function but where the usual truth values are replaced by sets, to be thought of sets of ways of realising truth. By modelling a process as a presheaf one allowed for the process possibly following several computation paths of the same shape and kept track of how the paths of the process branched nondeterministically. Presheaf models for concurrency connected concurrent computation with a rich mathematics, in particular the mathematics of species, but their operational reading could be challenging. Sometimes though, a denotational semantics in terms of presheaves could be represented by event structures; technically the category of elements of the presheaf denotation took the form of the configurations of an event structure. By bringing the role of event structures to the fore this eventually led to a game semantics based on event structures—Section 8.

6 French influence

Jean-Yves Girard has been an imposing figure in French logic and computation. He has a distrust of what he sees as too simple and over-arching use of algebra to structure and analyse logic. He has been exaggeratedly rude about Alfred Tarski's definition of truth in a model for first-order logic, and indeed about denotational semantics, to which his work has nevertheless contributed enormously.[2] Girard's work emphasises an operational understanding of proof and computation. This is far from saying it forsakes mathematical models, or concentrates on syntax in the way of traditional proof theory. On the contrary, the models he has developed and inspired have considerable ingenuity and depth, and have shifted interest to new ways of understanding proof and computation.

Through his reinvention of stable domain theory in the more restricted setting of coherence spaces, Girard was led to the important discovery of *linear logic* in the mid 1980's [21]. This gave a deconstruction of traditional logic into a more fundamental resource-conscious logic. That work helped turn the emphasis of domain theory away from function spaces supporting "currying" w.r.t. a product, to w.r.t. more general tensor products. In the jargon of category theory, it shifted the emphasis from cartesian-closed to monoidal-closed categories. Now in semantics of computation we see models of linear logic everywhere. Girard's coherence spaces correspond to a very special form of event structure in which causal dependency is the trivial identity relation.

In studying the proofs of linear logic, Girard discovered *geometry of interaction* (GoI) [38]. Although originally explained in terms of the mathematical structures of quantum mechanics, GoI was shown by Samson Abramsky and Radha Jagadeesan to have a more traditional, domain-theoretic reading in which the mechanism of interaction was that of least fixed points of domains in the manner of Kahn networks [39]. GoI was related to Jean-Jacques Lévy's optimal reduction of the lambda-calculus by Martin Abadi, George Gonthier and Lévy and has influenced the implementation of programming languages, notably via token-based computation [40]. Today GoI is perhaps most often viewed as an early form of game semantics—see Section 8.3.

7 Game semantics

There was some vagueness about what a solution to the full-abstraction problem for PCF entailed.

The 1980's had seen several technical successes, through the use of representations of domains: concrete data structures [18] and event structures [32] to give an operational description of how functions compute; information systems due to Scott and the author to give a logical presentation, and considerably simplify the recursive definition of domains and their logical relations [41, 42, 43]. Girard's work too, often exploited the fine structure of such representations.

[2]For instance in domain models of polymorphism [37] and through linear logic [21].

Given this history, it is not surprising that several early attempts to construct a fully-abstract model for PCF were based on adjoining extra structure to domains or their representations; for example using both the Scott order and stable order on functions in bidomains and bistructures [44], via Ehrhard's hypercoherences [45] or O'Hearn and Riecke's powerful logical relations [46]. These attempts were put paid to, or at least compromised, by Ralph Loader. In a tour de force Loader showed that the full-abstraction problem for PCF, as originally understood, couldn't be achieved effectively; the presentation of a fully abstract domain model for PCF would be non-computable [47].

This left open the intermediate question of whether there were other more independently motivated models in which all the finite elements were definable by PCF terms; from which then a (non-effective) domain model could be obtained by quotienting. To this question, called "intensional full-abstraction," two different affirmative answers were given and pioneered the highly informative use of games in the semantics of programming languages. Samson Abramsky, Radha Jagadeesan and Pasquale Malacaria invented AJM games [48], while Martin Hyland, Luke Ong, and independently Hanno Nickau, discovered HO games [49]. In many ways game semantics fitted the bill for a more operationally tuned domain theory; the role of domains was replaced by games and that of continuous functions by strategies. The role of games was extended beyond functional to imperative programs.

To give a flavour of game semantics of programs we give slightly nonstandard presentations of games and strategies as event structures; they are instances of concurrent games and strategies described in Section 8. We first describe the game generally associated with the type of Booleans as an event structure. In this game the first move is by Opponent (standing for the environment) asking for a value, the event $\boxminus$*, to which Player (standing for the program) may then respond by a move providing a value, either true represented by the event* $\boxplus_T$ *or false by the conflicting event* $\boxplus_F$*:*

$$\begin{tikzcd}
\boxplus_T \arrow[rr, no head, squiggly] & & \boxplus_F \\
& \boxminus \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

With an eye to illustrating type constructions on games, let's simplify the game. Imagine a type with a single value. We can represent this by a game in which Opponent's initial move is to ask for a value, the event $\boxminus$*, to which Player then may respond by a move providing the value, the event* $\boxplus$*. This game we draw as the event structure:*

$$\begin{tikzcd}
\boxplus \\
\boxminus \arrow[u]
\end{tikzcd}$$

Suppose we wish to represent the type of pairs of values as a game. We can do this simply by putting two copies of the single-value game in parallel, forming *event structure:*

$$\begin{tikzcd}
\boxplus & \boxplus \\
\boxminus \arrow[u] & \boxminus \arrow[u]
\end{tikzcd}$$

Opponent may ask for a value on the left and/or a value on the right, to which Player may or may not respond.

For the next step we ask how we can represent as a game the type of programs from pairs to a value, a form of function type. It's usual to consider a strategy in *a game. Less well-known but quite widely used is the idea of a strategy* from *one game* to *another.*[3] *According to this idea, explained a little further in Section 8, a strategy from the game of pairs to the single-value game is a strategy in the game below:*

$$\begin{tikzcd}
\boxminus & \boxminus & \boxplus \\
\boxplus \arrow[u] & \boxplus \arrow[u] & \boxminus \arrow[u]
\end{tikzcd}$$

The game is built as the parallel composition of the two games, but with the role of Player and Opponent reversed in the game for pairs. A strategy in this game is essentially a program. For example, consider the following event structure: it represents the strategy (or program) which first inspects input in the left component of a pair, then inspects the right component before yielding an output value.

$$\begin{tikzcd}
\boxminus \arrow[dr, dashed] & \boxminus \arrow[r, dashed] & \boxplus \\
\boxplus \arrow[u] & \boxplus \arrow[u] & \boxminus \arrow[u] \arrow[ll, dashed, bend left]
\end{tikzcd}$$

In general, a strategy in the game above, if it responds to the initial move of Opponent, will either output a value directly or demand more input from which to play further. I hope this gives an idea of how programs can be viewed as strategies. As here, a great deal of traditional game semantics is not concerned with winning conditions, though they have a role, for instance in ensuring a program is correct or progresses.

However the game story was far from complete. For one thing, it wasn't clear, at least initially, how to reconcile the two different versions, AJM and HO, of game semantics. For another, more significantly, the games were based on sequential plays in which Player and Opponent moves alternated.

The bias towards sequentiality has handicapped the theories of games in general. In *game theory* it has led to a menagerie of different kinds of games, each kind specialised to cope with one feature or another. There concurrency is handled in a piecemeal fashion, often through extra structure to capture imperfect information. This limits the ways that the games and strategies of

[3]The author first met this idea in a talk John Conway gave on Numbers and Games in the early 1970's to the Archimedeans—the mathematical society of the University of Cambridge.

game theory can be composed. While game semantics is very much concerned with structure and composition, its games are predominantly sequential; in most cases concurrency is represented indirectly via the interleaving of atomic actions of the participants. This rarely does justice to the distributed nature of the system described, inhibits analysis of its causal dependencies, and is often accompanied by compensatory, *ad hoc* fairness assumptions.

What was lacking was a rich algebraic theory of distributed/concurrent games in which Player and Opponent are more accurately thought of as teams of players, distributed over different locations, able to move and communicate. Although there are glimpses of such a theory in earlier work [50, 51, 52, 53, 54], a considerable unification occurs with the systematic use of event structures to formalise concurrent games and strategies through their causal structure [55, 56].

8 Concurrent strategies

Distributed/Concurrent games answer the need to rethink the foundations of games, to a more flexible grounding where they more truly belong, in the models and theories of interaction of computer science. The driving idea is to replace the role of games trees in more traditional developments by event structures.

A *concurrent game* is represented by an event structure together with a polarity marking its events to say whether they are moves of Player, marked + (the system) or Opponent, marked – (the unknown environment). Games often have extra features such as winning conditions, describing those configurations at which Player wins, or a payoff functions assigning a reward to each configuration. For simplicity, here we assume games are *race-free*, *i.e.* that there is no immediate conflict between a Player and an Opponent move; there may be conflict between Player and Opponent moves but it must be inherited from conflict between earlier moves, either both of Player, or both of Opponent.

With games as event structures the history of a play of a game is no longer described by a sequence of moves, but by a partial order expressing their causal dependency. The transition from total to partial order brings in its wake technical difficulties and potential for undue complexity unless it's done artfully. Fortunately one can harness the mathematical tools developed for interacting processes, specifically on event structures [32, 33].

There are two fundamentally important operations on two-party games. One is that of forming the *dual* game in which the roles of Player and Opponent are interchanged. On an event structure with polarity A this amounts to reversing the polarities of events to produce the dual $A^\perp$. By a strategy in a game we implicitly mean a strategy for Player. A strategy for Opponent, or counterstrategy, in a game A is identified with a strategy in $A^\perp$. The other operation is a *parallel composition* of games, achieved on event structures A and B by simply juxtaposing them, with events from different components, not in conflict, to form $A \| B$.

Following ideas of Conway and Joyal [57, 58], a strategy σ *from* a game A *to* a game B is taken to be a strategy *in* the compound game $A^\perp \| B$. Given another strategy τ from the game B to a game C the *composition* $\tau\odot\sigma$ is given essentially by playing the two strategies against each other over the common game B, and then hiding that interaction.

But what is a strategy in a concurrent game? We refer the reader to [55] for the detailed answer and its rationale: a definition of strategy is chosen precisely to make the copycat strategy the identity w.r.t. composition. But, roughly, a strategy in a concurrent game prescribes moves and dependencies of moves for Player which both obey the constraints of the game and do not constrain Opponent's behaviour beyond the constraints of the game. The axioms on strategies entail a formal connection with presheaf models mentioned earlier in Section 5 and through them with Scott domains [59]. In general strategies may be nondeterministic, but we can restrict to deterministic strategies in which all nondeterministic behaviour stems from Opponent.

As an example of a strategy, consider the *copycat* strategy in the game $A^\perp \| A$ which, following the spirit of a copycat, has Player copy the corresponding Opponent moves in the other component. The copycat strategy $\mathrm{C\!\!C}_A$ is obtained by adding extra causal dependencies to $A^\perp \| A$ so that any Player move in either component causally depends on its copy, an Opponent move, in the other component. It is illustrated below when A is the simple game comprising a Player move causally dependent on a single Opponent move:

$$\begin{tikzcd}
 & \boxminus \arrow[rr, dashed] & & \boxplus & \\
A^\perp & & \mathrm{C\!\!C}_A & & A \\
 & \boxplus \arrow[uu] & & \boxminus \arrow[ll, dashed] \arrow[uu] &
\end{tikzcd}$$

Strategies are not always obtained by simply adding extra causal dependencies to the game. In general, a strategy in a game A is expressed as a map of event structures $\sigma : S \to A$ describing the choices of Player moves by the event structure S. For example, consider the game comprising two Opponent moves in parallel with a Player move, and the strategy (for Player) in which Player makes their move if Opponent makes one of theirs. It is represented by the map

$$\begin{tikzcd}
\boxplus \arrow[r, no head, squiggly] & \boxplus & \arrow[r, "\sigma"] & & \boxplus & \\
\boxminus \arrow[u] & \boxminus \arrow[u] & & \boxminus & & \boxminus
\end{tikzcd}$$

which takes the two conflicting Player moves on the left to the single Player move on the right and each Opponent move on the left to the corresponding Opponent move on the right.

Not all maps of event structures $\sigma : S \to A$ are strategies. There are two further axioms on maps for them to be deemed strategies, receptivity *and* (linear) innocence. *Intuitively, they prevent Player from constraining Opponent's behaviour further than is allowed by the game. Receptivity expresses that any Opponent move allowed from a reachable position of the game is present as a possible move in the strategy. Innocence says a strategy can only adjoin new causal dependencies of the form $\boxminus \rightarrowtriangle \boxplus$, where Player awaits moves of Opponent, beyond those inherited from the game. Silvain Rideau and the author have* *shown that the axioms are precisely those that make copycat the identity for the composition of strategies [55].*

A strategy from *a game* A to *a game* B *is a strategy* in *the compound game* $A^\perp \| B$*; so a map* $\sigma : S \to A^\perp \| B$*. Given another strategy* $\tau : T \to B^\perp \| C$ *from* B *to a game* C*, the* composition $\tau \odot \sigma$ *is got by playing off the two strategies against each other over the game* B*. To do this precisely it is useful to harness two operations associated with maps of event structures:* pullback*, to produce the* interaction $\tau \circledast \sigma : T \circledast S \to A^\perp \| B \| C$*, which "synchronises" matching moves of* S *and* T *over the game* B*; then, a* partial-total factorisation property *of partial maps of event structures, to hide the synchronisations and produce, as its defined part, the strategy* composition $\tau \odot \sigma : T \odot S \to A^\perp \| C$*.*

A strategy $\sigma : S \to A$ *is* deterministic *if all conflict in* S *is inherited through causal dependency on conflicting Opponent moves. Deterministic strategies compose. That copycat is deterministic, so an identity for the composition of deterministic strategies, is due precisely to games being* race-free*. The strategy illustrated above is not deterministic.*

8.1 Winning conditions

Winning conditions of a game A specify a subset of its *winning configurations* W. An outcome in W is a win for Player. A strategy (for Player) is *winning* if it always prescribes moves for Player to end up in a winning configuration, no matter what the activity or inactivity of Opponent [60].

Formally, a strategy $\sigma : S \to A$ *is* winning *if* σx *is in* W *for all +-maximal configurations* x *of* S*; a configuration is +-maximal if no additional Player moves can occur from it. This can be shown equivalent to all plays of* σ *against counterstrategies of Opponent resulting in a win for Player.*

As the dual of a game A with winning conditions W we again reverse the roles of Player and Opponent to get $A^\perp$ and take its winning conditions to be the set-complement of W. In a simple parallel composition of games with winning conditions, $A \| B$, Player wins if they win in either component. With these extensions we can take a winning strategy from a game A to a game B, where both games have winning conditions, to be a winning strategy in the game $A^\perp \| B$. The choices ensure that the composition of winning strategies is winning. Because games are race-free, copycat will always be a winning strategy.

8.2 Imperfect information

In a game of *imperfect information* some moves are masked, or inaccessible, and strategies with dependencies on unseen moves are ruled out. One can extend games with imperfect information in a way that respects the operations of concurrent games and strategies [61]. Each move of a game is assigned a level in a global order of access levels; moves of the game or its strategies can only causally depend on moves at equal or lower levels.

In more detail, a fixed preorder of levels $(\Lambda, \leq)$ *is pre-supposed. A* Λ-game*, comprises a game A with a* level function $l : A \to \Lambda$ *such that if $a \leq_A a'$ then $l(a) \leq l(a')$ for all moves a, a' in A. A* Λ-strategy *in the Λ-game is a strategy $\sigma : S \to A$ for which if $s \leq_S s'$ then $l\sigma(s) \leq l\sigma(s')$ for all s, s' in S. The access levels of moves in a game are left undisturbed in forming the dual and parallel composition of games. As before a Λ-strategy from a Λ-game A to a Λ-game B is a Λ-strategy in the game $A^\perp \| B$. It can be shown that Λ-strategies compose.*

8.3 Old from new

The additional complexity of event structures over trees shouldn't obscure direct connections between strategies on concurrent games and the more familiar notions on games as trees. Event structures subsume trees. An event structure is *tree-like* when any two events are either in conflict or causally dependent, one on another; in this case its configurations form a tree w.r.t. inclusion, with root the empty configuration.

A *tree-like* game is one for which its underlying event structure is tree-like. Because we are assuming games are race-free, at any finite configuration of a tree-like game, the next moves, if there are any, are either purely those of Player, or purely those of Opponent; in this sense positions of a tree-like game either belong to Player or Opponent. At each position belonging to Player a deterministic strategy either chooses a unique move or to stay put. In contrast to many presentations of games, in a concurrent strategy Player isn't forced to make a move, though that can be encouraged through suitable winning conditions. Winning conditions specify those configurations at which Player wins, so in a tree-like game can be both finite and infinite branches in the tree of configurations.

Clearly the dual of a tree-like game is tree-like. A counterstrategy, as a strategy in the dual game, picks moves for Opponent at their configurations; when the counterstrategy is deterministic at each Opponent configuration it chooses to stay or make one particular move. As expected, the interaction of a deterministic strategy with a deterministic counterstrategy determines a finite or infinite branch in the tree of configurations, which in the presence of winning conditions will be designated as a win for one of the two players.

On tree-like games we recover familiar notions. More surprising is that by exploiting the richer structure of concurrent games we can recover other familiar paradigms, not traditionally tied to games, or if so only somewhat informally.

For example, by restricting to deterministic strategies between concurrent games where all moves are Player moves we rediscover *stable functions* and Berry's stable domain theory, of which Girard's qualitative domains and coherence spaces are special cases. For such restricted games, general, possibly nondeterministic, strategies correspond to *stable spans,* a model discovered and rediscovered in compositional accounts of nondeterministic dataflow.

In more detail, consider a strategy σ from one such purely Player game A to another B. This is a map $\sigma : S \to A^{\perp} \| B$ which is receptive and innocent. Notice that in $A^{\perp} \| B$ all the Opponent moves are in $A^{\perp}$ and all the Player moves are in B. By receptivity any configuration of A is copied as possible initial input in S. By innocence, the only new immediate causal connections, beyond those in $A^{\perp}$ and B, that can be introduced in S are those from Opponent moves of $A^{\perp}$ to a Player move in B. Beyond the causal dependencies inherited from the two games, S can only make a Player move in B causally depend on a finite subset of moves in $A^{\perp}$. For this reason any Player move s in S is associated with both an output event $out(s)$ in B and a demand on input $dem(s)$, a finite configuration of A needed for s to occur. A strategy between purely Player games corresponds to a stable span

$$\begin{tikzcd} & S^{+} \arrow[dl, "dem"'] \arrow[dr, "out"] & \\ A & & B \end{tikzcd} ,$$

involving two (special) functions from the event structure S^{+} obtained by restricting S to Player moves. For a general strategy, the output events for a particular input may be in conflict—the output is nondeterministic. When σ is deterministic, all conflicts are inherited from conflicts between Opponent moves. Then the strategy σ corresponds to a stable function *from the domain of configurations of A to the domain of configurations of B.*

Only marginally more complicated than those purely Player games are games which consist of two parallel components, one a purely Player game and the other with purely Opponent moves. The deterministic strategies between such games yield the Abramsky-Jagadeesan model for Geometry of Interaction built on stable functions [39].

Adjoining winning conditions and imperfect information to these games, so Opponent can see the moves of Player but not the converse, we recover a dialectica category [62], so Gödel's dialectica interpretation [63], from deterministic strategies. We obtain from Gödel's work an interpretation of proofs in first-order arithmetic as winning strategies. Dialectica categories, studied by Valeria de Paiva in her Cambridge PhD, mark an early occurrence of "lenses" used in functional programming, where they were invented independently to make composable local changes on data-structures [64, 65].

Generalising dialectica games a little they become examples of "container types" and deterministic strategies become "dependent lenses" [66]. Lenses generalise to "optics" in the context of general tensor products [67]. Optics based on stable spans are recovered when we allow the strategies to be nondeterministic.

Because concurrent games and strategies have been enriched, for example, to provide semantics to probabilistic and quantum programs [68, 69, 70, 71] these enrichments remain for these special cases. They show for example how to add probability to strategies between dialectica games. All this is dealt with in more detail in [4], and informally and more accessibly in [3].

The examples above concern ways of handling interaction within a functional approach. They have evolved, often independently. Any compositional theory of interaction is forced to handle the dichotomy between a system and its environment. Concurrent games and strategies address the dichotomy in fine detail, very locally, in a distributed fashion, through polarities on events. As the examples show, a functional approach has to handle the dichotomy much more ingeniously, through its cruder distinction between input and output; with basic interaction treated through the application of a function to its argument. Within concurrent games we can more clearly see what separates and connects the differing paradigms.

The examples don't cover all the ways in which functions are extended to cope with interaction. A notable omission in this article is the theory of effects in programming languages which uses the technology of monads and algebraic theories to refine the influential work of Eugenio Moggi and, roughly, describe computation in terms of enriched computation trees [72, 73]. Concurrent strategies have been enriched to address probabilistic and quantum computation. There is work to do in connecting concurrent games and strategies with the theory of effects.

9 Conclusion

Although this article discusses the limitations of domain theory, it also demonstrates its lasting power. Concurrent games and strategies provide a general model of interaction. Their generality can provide guidance in the form a model or its enrichment should take. In special cases they simplify to easier domain models. In one direction concurrent strategies help build domain models. In the other, when domain models are available they can simplify concurrent strategies enormously. It would be foolish to use a complicated model when a simpler one is available. In many contexts domain theory provides the simplest models we know!

Acknowledgements

My thanks to the anonymous reviewers for their helpful suggestions.

References

[1] Winskel, G.: The formal semantics of programming languages, an introduction. MIT Press. In Italian (UTET Libreria, 1999). In Chinese (China Machine Press, CITIC Publishing House, 2004). (1993)

[2] Abramsky, S., Jung, A.: Domain theory. Handbook of logic in computer science **3** (1994) 1–168

[3] Winskel, G.: Domain theory and interaction. Newsletter BCS-FACS FACTS Issue 2021-2 July (compressed) (2021)

[4] Winskel, G.: Making concurrency functional. CoRR **abs/2202.13910** (2022)

[5] Scott, D.S.: A type-theoretical alternative to ISWIM, CUCH, OWHY. Theor. Comput. Sci. **121**(1&2) (1993) 411–440

[6] Scott, D.S.: Mathematical concepts in programming language semantics. In: American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1972 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 16-18, 1972. Volume 40 of AFIPS Conference Proceedings., AFIPS (1972) 225–234

[7] Scott, D.S.: Data types as lattices. SIAM J. Comput. **5**(3) (1976) 522–587

[8] Milner, R.: Implementation and applications of scott's logic for computable functions. In: Proceedings of ACM Conference on Proving Assertions About Programs, Las Cruces, New Mexico, USA, January 6-7, 1972, ACM (1972) 1–6

[9] Gordon, M.J.C., Milner, R., Morris, L., Newey, M.C., Wadsworth, C.P.: A metalanguage for interactive proof in LCF. In Aho, A.V., Zilles, S.N., Szymanski, T.G., eds.: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, ACM Press (1978) 119–130

[10] Landin, P.J.: The next 700 programming languages. Commun. ACM **9**(3) (1966) 157–166

[11] Manna, Z., Vuillemin, J.: Fix point approach to the theory of computation. Commun. ACM **15**(7) (1972) 528–536

[12] Strachey, C.S., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. High. Order Symb. Comput. **13**(1/2) (2000) 135–152

[13] Plotkin, G.D.: A powerdomain construction. SIAM J. Comput. **5**(3) (1976) 452–487

[14] Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, J.L., ed.: Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974, North-Holland (1974) 471–475

[15] Brock, J.D., Ackerman, W.B.: Scenarios: A model of non-determinate computation. In Díaz, J., Ramos, I., eds.: Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19-25, 1981, Proceedings. Volume 107 of Lecture Notes in Computer Science., Springer (1981) 252–259

[16] Winskel, G.: Events, causality and symmetry. Comput. J. **54**(1) (2011) 42–57

[17] Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3) (1977) 223–255

[18] Kahn, G., Plotkin, G.D.: Concrete domains. Theor. Comput. Sci. **121**(1&2) (1993) 187–277

[19] Berry, G.: Stable models of typed lambda-calculi. In: ICALP. Volume 62 of Lecture Notes in Computer Science., Springer (1978) 72–89

[20] Girard, J.: The system F of variable types, fifteen years later. Theor. Comput. Sci. **45**(2) (1986) 159–192

[21] Girard, J.: Linear logic. Theor. Comput. Sci. **50** (1987) 1–102

[22] Berry, G., Curien, P.: Sequential algorithms on concrete data structures. Theor. Comput. Sci. **20** (1982) 265–321

[23] Curien, P.: On the symmetry of sequentiality. In Brookes, S.D., Main, M.G., Melton, A., Mislove, M.W., Schmidt, D.A., eds.: Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings. Volume 802 of Lecture Notes in Computer Science., Springer (1993) 29–71

[24] Rabinovich, A.M., Trakhtenbrot, B.A.: Communication among relations (extended abstract). In Paterson, M., ed.: Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings. Volume 443 of Lecture Notes in Computer Science., Springer (1990) 294–307

[25] Russell, J.: Full abstraction for nondeterministic dataflow networks. In: 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, Los Alamitos, CA, USA, IEEE Computer Society (nov 1989) 170–175

[26] Saunders-Evans, L., Winskel, G.: Event structure spans for nondeterministic dataflow. Electr. Notes Theor. Comput. Sci. 175(3): 109-129 (2007)

[27] Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer (1980)

[28] Brookes, S., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31** (1984) 560–599

[29] Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains. TCS **13** (1981) 85–108

[30] Winskel, G.: Events in computation. (1980) PhD thesis, University of Edinburgh.

[31] Winskel, G.: Event structure semantics for CCS and related languages. In: ICALP'82. Volume 140 of LNCS., Springer, A full version is available from Winskel's Cambridge homepage (1982)

[32] Winskel, G.: Event structures. In: Advances in Petri Nets. Volume 255 of LNCS., Springer (1986) 325–392

[33] Winskel, G., Nielsen, M.: Models for Concurrency. In: Handbook of Logic in Computer Science 4. OUP (1995) 1–148

[34] Joyal, A., Nielsen, M., Winskel, G.: Bisimulation from open maps. Inf. Comput. **127**(2) (1996) 164–185

[35] Hyland, M.: Some reasons for generalising domain theory. Mathematical Structures in Computer Science **20**(2) (2010) 239–265

[36] Cattani, G.L., Winskel, G.: Profunctors, open maps and bisimulation. Mathematical Structures in Computer Science **15**(3) (2005) 553–614

[37] Girard, J.: The system F of variable types, fifteen years later. Theor. Comput. Sci. **45**(2) (1986) 159–192

[38] Girard, J.: Towards a geometry of interaction. Contemporary Mathematics **92** (1989) 69–108

[39] Abramsky, S., Jagadeesan, R.: New foundations for the geometry of interaction. Inf. Comput. **111**(1) (1994) 53–119

[40] Gonthier, G., Abadi, M., Lévy, J.J.: The geometry of optimal lambda reduction. In: POPL '92. (1992)

[41] Scott, D.S.: Domains for denotational semantics. In Nielsen, M., Schmidt, E.M., eds.: Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings. Volume 140 of Lecture Notes in Computer Science., Springer (1982) 577–613

[42] Winskel, G., Larsen, K.G.: Using information systems to solve recursive domain equations effectively. In Kahn, G., MacQueen, D.B., Plotkin, G.D., eds.: Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings. Volume 173 of Lecture Notes in Computer Science., Springer (1984) 109–129

[43] Abramsky, S.: Domain theory in logical form. In: Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987, IEEE Computer Society (1987) 47–53

[44] Plotkin, G.D., Winskel, G.: Bistructures, bidomains and linear logic. In Abiteboul, S., Shamir, E., eds.: Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings. Volume 820 of Lecture Notes in Computer Science., Springer (1994) 352–363

[45] Ehrhard, T.: Hypercoherences: A strongly stable model of linear logic. Math. Struct. Comput. Sci. **3**(4) (1993) 365–385

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

[47] Loader, R.: Finitary PCF is not decidable. Theor. Comput. Sci. **266**(1-2) (2001) 341–364

[48] Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2): 409-470 (2000)

[49] Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. 163(2): 285-408 (2000)

[50] Abramsky, S., Melliès, P.A.: Concurrent games and full completeness. In: LICS '99, IEEE Computer Society (1999)

[51] Laird, J.: Game semantics for higher-order concurrency. In Arun-Kumar, S., Garg, N., eds.: FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings. Volume 4337 of Lecture Notes in Computer Science., Springer (2006) 417–428

[52] Melliès, P.A., Mimram, S.: Asynchronous games : innocence without alternation. In: CONCUR '07. Volume 4703 of LNCS., Springer (2007)

[53] Ghica, D.R., Murawski, A.S.: Angelic semantics of fine-grained concurrency. In: FOSSACS'04, LNCS 2987, Springer (2004)

[54] Faggian, C., Piccolo, M.: Partial orders, event structures and linear strategies. In: TLCA '09. Volume 5608 of LNCS., Springer (2009)

[55] Rideau, S., Winskel, G.: Concurrent strategies. In: LICS 2011. (2011)

[56] Winskel, G.: ECSYM Notes: Event Structures, Stable Families and Concurrent Games. http://www.cl.cam.ac.uk/~gw104/ecsym-notes.pdf (2016)

[57] Conway, J.: On Numbers and Games. Wellesley, MA: A K Peters (2000)

[58] Joyal, A.: Remarques sur la théorie des jeux à deux personnes. Gazette des sciences mathématiques du Québec, 1(4) (1997)

[59] Winskel, G.: Strategies as profunctors. In: FOSSACS 2013. Lecture Notes in Computer Science, Springer (2013)

[60] Clairambault, P., Gutierrez, J., Winskel, G.: The winning ways of concurrent games. In: LICS 2012: 235-244. (2012)

[61] Winskel, G.: Winning, losing and drawing in concurrent games with perfect or imperfect information. In: Festschrift for Dexter Kozen. Volume 7230 of LNCS., Springer (2012)

[62] de Paiva, V.: The Dialectica categories. PhD Thesis, University of Cambridge (1988)

[63] Avigad, J., Feferman, S.: Gödel's functional (”dialectica”) interpretation. (1999) 337–405

[64] Oles, F.J.: A category theoretic approach to the semantics of programming languages. PhD Thesis, University of Syracuse (1982)

[65] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007) 17

[66] Abbott, M.G., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theor. Comput. Sci. **342**(1) (2005) 3–27

[67] Pickering, M., Gibbons, J., Wu, N.: Profunctor optics: Modular data accessors. Art Sci. Eng. Program. **1**(2) (2017) 7

[68] Winskel, G.: Distributed probabilistic and quantum strategies. Electr. Notes Theor. Comput. Sci. 298: 403-425 (2013)

[69] Paquet, H., Winskel, G.: Continuous probability distributions in concurrent games. In Staton, S., ed.: Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018. Volume 341 of Electronic Notes in Theoretical Computer Science., Elsevier (2018) 321–344

[70] Clairambault, P., de Visme, M., Winskel, G.: Game semantics for quantum programming. Proc. ACM Program. Lang. **3**(POPL) (2019) 32:1–32:29

[71] Clairambault, P., de Visme, M.: Full abstraction for the quantum lambda-calculus. Proc. ACM Program. Lang. **4**(POPL) (2020) 63:1–63:28

[72] Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989, IEEE Computer Society (1989) 14–23

[73] Plotkin, G.D., Power, A.J.: Computational effects and operations: An overview. Electron. Notes Theor. Comput. Sci. **73** (2004) 149–163