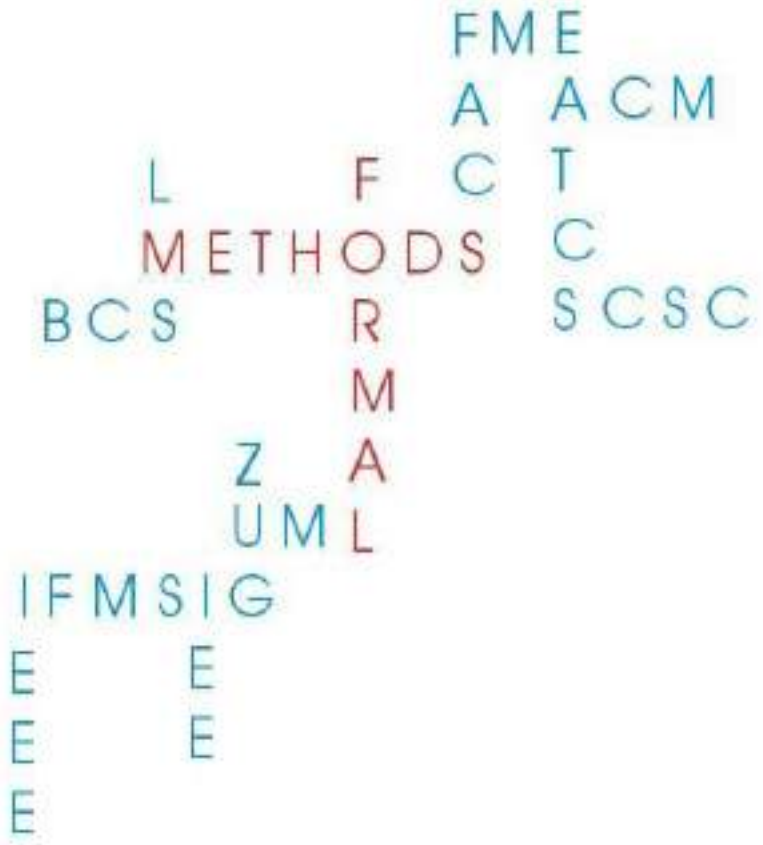
Issue 2021-2
July 2021

FACS FACTS



Formal Aspects of Computing Science Specialist Group

The Newsletter of the Formal Aspects of Computing Science (FACS) Specialist Group

ISSN 0950-1231

About FACS FACTS

FACS FACTS (ISSN: 0950-1231) is the newsletter of the BCS Specialist Group on Formal Aspects of Computing Science (FACS). FACS FACTS is distributed in electronic form to all FACS members.

Submissions to FACS FACTS are always welcome. Please visit the newsletter area of the BCS FACS website for further details at:

https://www.bcs.org/membership/member-communities/facs-formal-aspects-of-computing-science-group/newsletters/

Back issues of FACS FACTS are available for download from:

https://www.bcs.org/membership/member-communities/facs-formal-aspects-of-computing-science-group/newsletters/back-issues-of-facs-facts/

The FACS FACTS Team

Newsletter Editors

Tim Denvir timdenvir@bcs.org
Brian Monahan brianqmonahan@googlemail.com

Contributors to this issue:

Jonathan Bowen, Andrew Johnstone, Keith Lines,
Brian Monahan, John Tucker, Glynn Winskel

BCS-FACS websites

BCS: http://www.bcs-facs.org
LinkedIn: https://www.linkedin.com/groups/2427579/
Facebook: http://www.facebook.com/pages/BCS-FACS/120243984688255
Wikipedia: http://en.wikipedia.org/wiki/BCS-FACS

If you have any questions about BCS-FACS, please send these to Jonathan Bowen at jonathan.bowen@lsbu.ac.uk.

Editorial

Dear readers,

Welcome to the 2021-2 issue of the FACS FACTS Newsletter. A theme for this issue is suggested by the thought that it is just over 50 years since the birth of Domain Theory[1].

Why did computer science need Domain Theory? To provide a semantics for computational structures, one needs to construct a mathematical model for them. In many high-level programming languages it is possible to define recursive data types, and to write successful programs which use them. Yet it is impossible to model recursive data types, such as those that contain their own function spaces, in set theory. Georg Cantor showed this in about 1874-1884[2]. Where data types embody functions, they are computable functions, not the fully fledged functions found in set theory. Domains provide a way of expressing these limited functions, which has, almost happily, the accidental effect of enabling types to have this kind of recursion.

Programming languages were not the first context in which we find recursion. Developments in mathematical logic, in the early 20th century, used the idea decades before. In their introduction to the second edition of *Principia Mathematica*[3] Whitehead and Russell refer to that theorem of Cantor's, stating it very succinctly:

$$2^n > n$$

Here 2 denotes the two-value set $\{0,1\}$, n denotes the set of natural numbers $[0:n-1]$, 2^n denotes the set of functions from n to 2 and $>$ compares the cardinalities of the sets. Whitehead and Russell remark that Cantor's proof is limited to finite sets n, but if that historically is the case, it is easy to extend a natural proof to infinite sets of any cardinality (left to the reader!). (Note that if n is countably infinite, then the theorem and proof are isomorphic to another strongly related theorem of Cantor's, that the Reals are uncountable).

Enough of this from me. Our first feature article is by my co-editor, Brian Monahan, *Domain Theory Revisited*, an introduction and thoughtful discourse on the subject. Then comes John Tucker, *Haskell B. Curry at War*, a historical note from the History of Computing Collection at Swansea University. The third feature is by Glynn Winskel, *Domain Theory and Interaction*. This is a splendid grand tour of the history from Domain Theory's beginnings, through interactive computation, concurrent games and

[1] Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG-2, Oxford University Computing Laboratory, Oxford, England, November 1970; Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages Oxford Programming Research Group Technical Monograph. PRG-6. 1971.

[2] See e.g. https://mathshistory.st-andrews.ac.uk/Biographies/Cantor/

[3] Alfred North Whitehead & Bertrand Russell, *Principia Mathematica*, CUP 1910, second edition 1927, paperback edition to *56 1962, page xiv.

strategies, to recent work on dialectica categories and container types. Then, Andrew Johnstone provides a most instructive review of the recent book by Cliff Jones, *Understanding Programming Languages* (Springer Switzerland 2020), again extremely relevant to program language semantics.

Finally we have reports on some recent FACS and FACS-related events. Keith Lines summarises the webinar by Conor McBride & Fredrik Nordvall Forsberg, *Dimensionally correct by construction: Type systems for programs*. Jonathan Bowen details the ABZ 2021 conference, which was preceded by a festschrift for Egon Börger. Jonathan also covers FACS and other relevant seminars that have taken place so far this year: Keith Lines, *NPL's Experience with Formal Aspects*; Marta Kwiatkowska, *Probabilistic Model Checking for the Data-rich World* (BCS Lovelace seminar); and Michael Leuschel, *New Ways of Using Formal Models in Industry* (joint FACS and FME seminar).

We hope you enjoy FACS FACTS issue 2021-2.

Tim Denvir
Brian Monahan

Domain Theory – Revisited

Brian Monahan

Introduction

Just over 50 years ago, the logician Dana Scott discovered a way to give an elegant *mathematical semantics* to Church's (untyped) lambda calculus. At the time, this was an extraordinary and most unexpected discovery that required some surprising and unanticipated developments in the kind of mathematics involved.

This mathematics made significant use of a range of concepts, most of which were entirely new to Theoretical Computer Science, at least up until that point. Although this initially seemed all rather shocking and strange – certainly unusual – the mathematics came to be seen as the beginning of a liberating revolution with some far reaching consequences. The mathematical framework in question was, of course, *Domain Theory*, as it later came to be known.

What is perhaps difficult for us to see today is just how controversial and infamous lambda calculus had become amongst logicians and theoreticians of the day. From the time it emerged in the 1920s as a *primitive* notion of "function as rules", lambda calculus was quickly seen by logicians to be rather problematical, despite consisting of only a few equational rules. It was regarded with much suspicion and considerable doubt as to its utility and consistency, since it was plagued with seeming foundational questions from the outset, such as finite expressions which appeared to never terminate, and particular terms which required arbitrary functions to have fix-points – and so on. Perhaps the most shocking thing of all was that the lambda calculus was envisaged as modelling something quite basic and fundamental – the primitive idea of "functions as rules" – and yet it proved to be far from simple! If *this* could be so problematical, what hope could there be for tackling anything more complicated?

The distinction between the *typed* and *untyped* lambda calculus is very important here. The simply typed lambda calculus seemed to be far better behaved and relatively tame (due to strong normalization), compared to its apparently unruly sibling, the untyped lambda calculus. However, this only led to yet more unsettling questions and creating even greater unease and suspicion in the minds of logicians. Given the simplicity, what could possibly have *gone wrong*? The answer was that *nothing* had gone wrong – we just hadn't yet understood what the rules *described*.

Roadmap

This essay takes an informal tour through various issues surrounding Domain Theory, such as:

- What were the underlying questions that motivated Domain Theory's development?
- What problems can Domain Theory help us solve?
- How does Domain Theory work, mathematically speaking?
- And, what insights were needed?

The tour concludes by examining what Domain Theory opened up for Computing Science and where it has led us to today.

What this essay *doesn't* set out to do is to present a definitive historical account of the development of Domain Theory – the story presented here is more concerned with the ideas themselves and their inter-relationships. One may already find a very extensive historical account of the origins and development of both lambda calculus and Domain Theory from Felice Cardone and Roger Hindley [7].

My primary source for technical understanding, inspiration and background has been an extensive chapter from the *Handbook of Logic in Computer Science* called *Domain Theory*, by Samson Abramsky and Achim Jung [2]. It has proved to be an invaluable and reliable resource concerning the mathematics of Domain Theory.

Logic, Language and Programming

These days we all know what programming is - programming involves instructing a device to perform a series of calculations that solves some problem by producing outputs, given suitable inputs. These instructions determine the behaviour of the device and they may be given in the form of textual computer programs, machine instructions – or even simply by selecting and clicking on something in a browser window with a mouse.

The issue urgently confronting computing in the late 1960s and early 1970s was to do with organising and instructing computer systems on an ever-increasing scale. It was clear that this would need to be done through the use of computer programs written in a variety of computer programming languages and later translated into machine instructions in an automated manner.

However, it was soon realised that the problem at hand was *two-fold* in nature;

- Firstly, it concerned how to keep programs manageable so that precise instructions for behaviour could be provided in machine format - via machine translators and compilers.

- Secondly, how to make programs sufficiently intelligible to people so that an accurate human understanding of the machine's behaviour given by the program could be obtained purely from the program text itself. Without this, it is very doubtful how anyone[1] could organise, marshall and be confident of what any given program *would* do – or even what it *could* do.

The issue became one of *communication* between human and machine in two distinct senses – providing *precise instructions* to the machine to direct its activity, and *human understanding* for creating and reasoning about systems behaviour. In short, programs should provide the basis for *reasoning* about the behaviour they precisely describe. Both logic and language are required for effective programming.

The remainder of this essay is concerned with the basics of what Domain Theory itself does and how its application helped to make programming itself more reasonable and therefore more effective. This story involves understanding the primitive basis of programming in terms of the (untyped) lambda calculus and showing how this primitive calculus was given a precise mathematical semantics, despite the calculus itself involving self-reference, thereby courting potential circularity and inconsistency.

What problem does semantics help solve?

Why do we need (model-theoretic) mathematical semantics – what problems can it help us solve?

The short answer is that semantics helps to show that what we say is capable of being meaningful, of conveying meaning. Imagine we are doing formal reasoning of some kind in, say, something like first order predicate logic. What we would like to know is that, when our formal reasoning is well-formed, we can be sure that the inferences we make produce, at each stage, statements that *make sense* (i.e. they are *valid*).

To do this, we firstly need to introduce a mathematical semantics as a function that can assign a suitable *value* to every well-formed expression and relational formula – we say that such a value is a *denotation*. Typically this semantic function will need to depend upon an *interpretation* (or *model*) associating denotations with primitive symbols and terms of the language. Furthermore, we will say that any semantic function associat-

[1] Of course, where 'anyone' here includes *oneself*, after having written and tested the code only six months previously!

ing denotations with expressions and formulae (usually defined by induction over the syntactical phrase structure), is known as a *denotational semantics*.

Formal semantics then defines what validity means by introducing a *logical consequence* (or *satisfaction*) relation over the structure of formulae. This says what it means for universally closed statements (or *sentences* having no free variables) to be *valid*, relative to a particular model. Sentences are then said to be (universally) *valid* whenever they are valid with respect to all (appropriate) models. We also say that a system of inference rules is *sound* whenever each inference rule preserves validity – that is, whenever the premisses are valid, then so is the conclusion. Sound inferences map valid statements into valid statements.

Obviously, the soundness of a logic or calculus is a very desirable property to have – indeed, not possessing soundness would make it *worse* than useless!

Overall, semantics is generally needed to say what is meant by soundness of inference. Without a semantics and a consequential notion of soundness, it calls into question whether deduction and inference is even meaningful, and potentially undermining confidence in the utility of the entire calculus.

It's worth explicitly saying that formal calculi are often symbolically defined in a purely algebraic style just in terms of equations or proof-theoretically in terms of inference rules (e.g. process algebra, Structural Operational Semantics). This might be done without any particular or overriding concern for providing a model-based denotational semantics. In these cases, the focus of concern naturally becomes one of *formal derivability* using the inference rules. The extent to which that has any significance or not boils down to what the calculus in question is for. Denotational semantics generally makes an appearance to provide a mathematical correspondence between logical statements *about* things on the one hand, and those relationships that hold *between* things on the other.

Consistency

The question of *consistency* is related to the notion of validity – and is therefore right at the core of our concerns. To discuss this, it is perhaps easier to say what *lack* of consistency, or *inconsistency*, means for a logic or calculus. Having done this, we then define *consistency* to mean the *lack of inconsistency*, or equivalently, *not inconsistent*.

The general idea is that inconsistent systems derive *absurd* conclusions - such as asserting that all values are the same when there are at least two distinct values (e.g. $0 = 1$). Inconsistency means that validity collapses, leading to all propositions being identified together – making truthhood and falsehood indistinguishable. We may more generally say that a logic is *inconsistent* when every possible sentence or statement is derivable –

and therefore no *distinctions* could be drawn by using inferences in that system.

From a model-theoretic point of view, this immediately says that there cannot be any well-defined, non-trivial models for an inconsistent logical system – because in any non-trivial model, only some propositions will hold, and others will not. Since inconsistency collapses validity and all propositions would be identified together, there could not then be any non-trivial models.

Consequently, a consistent system always has at least one well-defined, non-trivial model (typically many, many more than one!) – and, from an inference point of view, not every statement or equation is derivable.

All of this will have particular significance when we come to consider the semantics of the untyped lambda calculus below.

Untyped Lambda Calculus

The untyped lambda calculus arose in the late 1920s and was pioneered by a number of logicians, primarily Alonzo Church and then Haskell Curry. This formal system provided a way to explore functional forms involving abstraction, substitution and application [13, 4]. The (simply) typed lambda calculus came about somewhat later in the early 1930s. For completeness, here are the basic rules of the untyped lambda calculus[2]:

α **conv:**	$\lambda x \cdot M \equiv \lambda y \cdot M[x := y]$	(for variable y not occurring freely in M)
β **conv:**	$(\lambda x \cdot M)\ N \equiv M[x := N]$	
η **conv:**	$(\lambda x \cdot M(x)) \equiv M$	(for variable x not occurring freely in M)

The above are generally regarded as left-to-right *reduction rules*. The α reduction rule only renames bound variables within terms, bringing no change. On the other hand, β reduction 'unfolds' the terms on the left, therefore doing something more radical. Finally, η reduction cancels redundant lambda's.

After applying these rules in a (possibly empty) sequence of reductions, a term is said to be in *normal-form* (or *normalised*) when no further β or η reductions are possible, modulo α conversion.

However, this begged some questions. For example, does it matter how each of these strikingly simple rules is applied? Could *different* normal-forms be produced by different reduction sequences?

These questions were cleared up by Alonzo Church and J. Barkley Rosser in 1936 when

[2]The substitution notation $M[x := N]$ means "substitute term N for all free occurrences of x within term M".

they showed that reduction[3] is *confluent*. This meant that for any term t that reduces to term t_1 and also reduces to term t_2, then there exists a common term t_3 where both terms t_1 and t_2 reduce to t_3 [15]. Furthermore, this implies that normal-forms are *unique*, since if any reduction sequence could produce a term in normal-form, then all of them could, and what's more, each of these normal-forms have to be equal to each other, by transitivity of equality.

This removed some degree of concern for the untyped lambda calculus. When terms possessed normal-forms, then reduction sequences behaved in a deterministic manner, yielding a single result. However, as illustrated below, not all reduction sequences normalised, raising further concern over what the untyped lambda calculus might mean.

Self-Reference, Circularity ... and Non-Termination

At the core of this issue are those *meaningful* lambda terms that do not reduce to normal-form, such as, for example, Curry's fix-point combinator, Y:

$$Y \equiv \lambda f \cdot (\lambda x \cdot (x\ (x)))(\lambda y \cdot f(y\ (y)))$$

Applications of this expression do not normalise because of the following derivation[4]:

$$\begin{aligned}
Y(g) &\equiv \underline{(\lambda x \cdot (x\ (x)))}(\lambda y \cdot g(y\ (y))) && \text{[1] – defn. of } Y + \beta \text{ conv.} \\
&\equiv \underline{(\lambda z \cdot g(z\ (z)))}(\lambda y \cdot g(y\ (y))) && \text{[2] – } \beta \text{ conv. + } \alpha \text{ conv.} \\
&\equiv g(\underline{(\lambda z \cdot g(z\ (z)))(\lambda y \cdot g(y\ (y)))}) && \text{[3] – } \beta \text{ conv. + } \alpha \text{ conv.} \\
&\equiv g(\underline{Y(g)}) && \text{[4] – back substitution from [2]} \\
&\equiv g(g(g(\cdots))) && \text{[5] – by inductive repetition of [4]}
\end{aligned}$$

The above well-formed expression for Y contains *self-applications* such as $(x\ (x))$. Self-applications like this illustrates how every entity could always *receive* arguments and may also be *applied* as an argument, even to itself. As can be seen, the self-application directly gives rise to a circularity at each stage, where a fresh application is produced. This is effectively a direct, unguarded loop – and, as we may all appreciate, this leads to non-terminating reduction behaviour when evaluated eagerly.

Combinatory Logic (CL)

In an earlier and parallel development to untyped lambda calculus, Combinatory Logic (or CL) was formulated and explored originally by Moses Schöenfinkel in 1924, and

[3] Originally proven for a more restricted variant of untyped lambda calculus.
[4] The underlining indicates the site of substitution – typically β conversion.

then rediscovered by Haskell Curry. The general idea was to develop a way of expressing functional forms, but without the *explicit* use of bound variables. In modern terms, this amounts to an equational approach that expresses combinatory functions, in much the same way that untyped lambda calculus does [13, 4, 28]. We have:

$$\text{S}\ \vdash S\ f\ g\ x\ \equiv\ f\ x\ (g\ x) \qquad \text{K}\ \vdash K\ x\ y\ \equiv\ x$$

$$\text{eq}\ \vdash x\ \equiv\ x \qquad \text{sym}\ \frac{\vdash x\ \equiv\ y}{\vdash y\ \equiv\ x}$$

$$\text{trans}\ \frac{\vdash x\ \equiv\ y \qquad \vdash y\ \equiv\ z}{\vdash x\ \equiv\ z}$$

$$\text{apply}\ \frac{\vdash x\ \equiv\ y}{\vdash f\ x\ \equiv\ f\ y} \qquad \text{fun-eq}\ \frac{\vdash f\ \equiv\ g}{\vdash f\ x\ \equiv\ g\ x}$$

Equational inference rules for {S,K} Combinatory Logic

There is an elegant simplicity and a purity of form in CL. Whereas the rules for the untyped lambda calculus involves a meta-logical "substitution" notation, no such device is necessary in formalising CL. Because these notations are inter-convertible and therefore expressively equivalent, the issues inherent in either form necessarily find expression in the other. Because of its spartan simplicity, expressions in CL are typically considerably longer than in pure lambda calculus, with a worst-case expansion of $O(n^3)$ [16]. For example, given below is the shortest possible SK term [30] representing a fix-point combinator – of which there are many:

$$SSK(S(K(SS(S(SSK))))K)$$

The particular combinator names, S and K have some informal mnemonic value – the S combinator can be thought of as a kind of '*S*ubstitution' operator, whereas the K combinator might be thought of as a kind of '*K*onstant' operator!

The core problem

The issue of course is that both CL and untyped lambda calculus were each very basic formal systems encoding functional forms together with a process of calculation, given in terms of equational reduction. It is hard to imagine a simpler system than these, that expressed the same intent.

Given this stark simplicity, it was intuitively clear that anything more practical or more involved in terms of computation would almost certainly exhibit aspects similar to those witnessed in these calculi. Therefore, studying these rather primitive systems seemed like a necessary step to make. And so it proves.

The general näive idea was that lambda terms formalised functions *intensionally* by defining them via equational rules, as opposed to defining them *extensionally* in terms of sets of ordered pairs. It was quite naturally assumed that these two notions exactly corresponded and were merely two different ways of describing the same thing.

However, as understanding of lambda calculus grew, it became increasingly clear that these two notions of function were not completely alike. In particular, because of the Y combinator, this showed that *all* lambda terms possess a fix-point – namely, the term $Y\ f$ since:

$$Y\ (f)\ \equiv\ f(Y\ (f))$$

Moreover, this also meant there were also terms that *failed* to normalize. This failure might be naturally accounted for by considering lambda terms just to represent *partial* functions – where the lack of normalization then corresponded to undefined behaviour.

The confluence result of Church-Rosser was especially welcome as it showed that the way lambda terms could be reduced to normal-form was entirely consistent with this suggestion (c.f. deterministic behaviour). Importantly, it also demonstrated that the lambda calculus was, from a proof-theoretic standpoint, consistent (e.g. not all values were equivalent).

However, there was a remaining serious issue, though: *every* lambda term represented a function-like entity, since every term could be applied to every term, including itself. This severely complicates the idea that lambda terms somehow represented simple functions of any kind. The difficulty here amounts to finding a non-trivial mathematical structure D *isomorphic* to its *own* set of functions – that is:

$$D\ \cong\ (D \to D)$$

However, this flatly contradicts Cantor's Theorem concerning cardinalities of sets and their corresponding sets of functions. The only possible set-theoretical solution might conceivably be to make D a trivial, one-point set and then the set of functions would then have to correspond to the set of *total* functions from D to D, just to satisfy the cardinality constraint[5].

However, using total functions here could not explain the lack of normalisation for some terms – nor the fact that the Church-Rosser result implies that D would have at least two distinct values. Problematical indeed!

[5] Any set of partial functions for any non-empty set has at least *two* elements.

All of the above was widely understood during the 1930s. Despite the Church-Rosser result clearly demonstrating consistency for the untyped lambda calculus as a pure reduction theory of functional forms, it was hard to reconcile this with what was generally understood mathematically about functions.

Unsurprisingly, this led to a general disquiet about the untyped lambda calculus amongst logicians and mathematicians. Of course, no such concerns plagued the simply-typed lambda calculus and typed combinatory logic – and generally speaking, research focus shifted towards those approaches instead.

Alan Turing and Lambda Calculus

One may therefore be forgiven for thinking that all this fuss about a somewhat bizarre equational calculus is rather blown out of proportion – after all, does it *really* matter if there were all these issues? The short answer is, yes, of course it does.

The basic reason is that, as part of Alan Turing's landmark work in 1936-1937 [31, 32, 23, 5] Turing showed that a number of what we would now call *computational models* (including lambda calculus) were all equivalent, in the sense that they were *interconvertible* – everything that could be expressed in one model, could also be expressed in any of the other models. What's more, Turing was able to present a convincing justification [31] that "effective computation" corresponds to performing symbolic operations according to rules (e.g. Turing Machines, Post Correspondence Problem, and Lambda Calculus) [8, 11].

This meant that untyped lambda calculus embodies full-blown *universal* computation in all its intricate variety – which provided even greater impetus to understand its nature. By giving a model-based semantics to untyped lambda calculus, it meant that these concerns of logicians could then be *resolved* in purely mathematical terms.

Although the Church-Rosser theorem had already shown in the 1930s that untyped lambda calculus is consistent proof-theoretically, what wasn't known until Scott was if a model-based semantics of untyped lambda calculus properly existed. Although it is now known that there are many *possible* models for the lambda calculus (consider Cartesian-Closed Categories), Domain Theory shows us that it is *consistent* to say:

- As intended, lambda terms denote functions of some description.
- Although not every term has a normal-form, even those terms that don't normalize are still meaningful and nevertheless possess a denotation. What's more, even though a sub-term may not be normalizing, a term containing such a term may still normalize overall.
- Elegantly explains why all lambda terms possess fix-points.

Properties of calculations in lambda calculus could be established just by mathematically examining and exploring the model itself. In this way, the model theory naturally provides a *metatheory* for the underlying calculus and its derivation structure.

The need for programming language semantics

This *puzzle* of the untyped lambda calculus languished for many years. Here was what appeared to be a remarkably simple calculus consisting of only a few rules that was intended to capture the basic idea of "function as rule" (i.e. computation) – and yet there were several deep issues concerning what it might mean!

During the 1960s and 1970s, the need to use programming languages to express programs and then to compile, translate and run them, became ever more pressing. Back then, most of the R&D effort on language development and design had focused almost exclusively on the 'parsing' problem – how to parse complex textual phrase structures into tree representations.

However, as languages become ever more sophisticated and therefore complex, the compilers, translators and other infrastructure necessarily become much more technically complicated and difficult to manage. For example, landmark languages like ALGOL 60 had by then emerged, as well as executable application languages like SIMULA, both of which needed complex processing to translate from phrase structures into machine instruction sequences. The issue of compiler correctness, and how to describe what that might mean, loomed large.

For example, the language SIMULA illustrates this trend of increasing tools sophistication arising from complex application needs that were provided for through language design. From the outset, SIMULA was considered mostly as a way to precisely describe complex arrangements of systems that interact to produce behaviour. Operationally, this included all sorts of complex structures, such as coroutines, discrete event simulation, garbage collection and of course, object-oriented programming in terms of objects and classes [17, 9].

At around this time, the idea of turning to logic and using some kind of mathematical approach was starting to emerge among researchers, in particular Christopher Strachey at the University of Oxford; this approach would involve extending and developing linguistics and logical structure to give meaning and content to programming languages. The whole area cried out for finding a way to utilise mathematics somehow to help bring some coherence and much needed structure to the general understanding of programming languages.

However, given the puzzle of the untyped lambda calculus – still unsolved at that time – it was clear that such a programme of research was never going to be straightforward.

Fortunately, Christopher Strachey was by then collaborating with Dana Scott, both of whom were no doubt very well-aware of the issues surrounding formal mathematical semantics. In particular, these concerns included the potentially problematical nature of general recursive definitions over syntactic phrase structures (See [20]).

The puzzle of the untyped lambda calculus mentioned earlier suggested that a novel approach to mathematical semantics of programming languages was required. This was the same kind of problem in each case after all – the untyped lambda calculus was known to be centrally concerned with computation, and it also seemed highly plausible that the foundational issues arising for providing the lambda calculus with a semantics could easily reappear when dealing with more complex programming languages.

This all pointed towards finding some degree of deep understanding of the lambda calculus itself and the mathematical difficulties of giving it a formal semantics.

The puzzle of the lambda calculus needed to be solved!

Scott's solution

The key insight was to realise that lambda terms weren't representing arbitrary functions between plain old sets of values, but instead they represented some kind of function over sets that were *endowed* with some extra mathematical structure. Furthurmore, these lambda definable functions were not entirely arbitrary themselves – they also *preserved* this additional structure.

Setting up the framework: Domains

In his 1970 PRG monograph "Outline of a Mathematical Theory of Computation" [19], Scott showed how a *semantic model* of the untyped lambda calculus could be constructed. The first part of the monograph introduced and motivated his framework underpinning his approach. Within this section, Scott in fact used the term "data type" to stand for what we might now call a domain[6]. To avoid any further confusion, the term 'domain' is used here instead. The framework Scott originally presented was given in terms of 5 'axioms' or required properties of domains and were as follows:

1. **A *domain* is a *partially ordered set*.**
   Domains form basic set-like entities and contain the elements or values of in-

[6]Within the practice of semantics, the term 'domain' is somewhat fluidly defined in general, simply because of the great variety of possible structures available for use. Typically, what 'domain' may mean is only more sharply defined within a particular context. In any event, domains are generally some class of partially ordered sets with additional structure and satisfying some particular constraints e.g. completeness.

terest. These domains are endowed with a partial ordering over the elements and represents approximation[7] between values, written as $x \sqsubseteq y$.

Intuitively, this ordering qualitatively captures how data can *approximate* other data – for example, consider a sequence of real numbers of greater and greater precision converging towards a real number *in the limit*.

2. **Mappings between domains are *monotonic*.**
The functions of interest (or mappings) all preserve this *ordering* on each domain. If $m : A \to B$ then, for all $a, a' : A \bullet a \sqsubseteq a'$ implies that $m(a) \sqsubseteq m(a')$.

Intuitively, monotonic mappings preserve approximations – as the inputs become better known, so also do the resulting outputs.

3. **A domain is a *complete lattice* under it's partial ordering.**
A partially ordered set P is a *complete lattice* whenever every subset S of P has a *least upper bound*, written as $\bigsqcup S$ in P. This also implies that every subset has a *greatest lower bound*, written as $\bigsqcap S$.

Taking the entire set P, there is necessarily a *least element* of P, written as $\bot$, as well as a *greatest element*, written as $\top$.

Intuitively, the lattice structure captures how information may be combined together to yield *improved* approximations. Not all combinations are consistent in that way and these yield the *over-determined* or *inconsistent* element, represented by $\top$, the greatest element of a domain.

4. **Mappings between domains are *continuous*.**
At this point, Scott brings in the key idea of *continuous function*, from topology, albeit in a unexpected way. The broad idea is that continuous functions preserve *limits of directed subsets*.

To see what's going on here, we need the idea of a *directed set* in a partially ordered set P. A non-empty subset D of P is a *directed subset* whenever for any two elements $a, b \in D$ there exists $d \in D$ such that $a \sqsubseteq d$ and $b \sqsubseteq d$.

We mention that monotonic functions *preserve* directed sets – that is, they map directed subsets for a domain into directed subsets for the result domain. This holds because monotonic functions preserve the relevant ordering.

However, in general, monotonic functions may or may *not* preserve limits – that is, least upper bounds of (infinite) directed sets. Because of this, we need to consider instead those functions which do.

[7] To reduce notational clutter, we do not use decoration to distinguish the orderings in different, but related, domains

We say that a monotonic function $f : A \to B$ is a *Scott-continuous* function whenever, for every directed subset C of A, we have that:

$$f(\bigsqcup C) = \bigsqcup \{\ f(c) \in B \mid c \in C\ \} = \bigsqcup f(C)$$

We now write $[A \to B]$ for the set of all Scott-continuous functions between domains A and B.

The intuition is that directed subsets, D of a domain T represent *consistent sets of approximations* of some value, $v = \bigsqcup D$ from T. The Scott continuous functions are then those functions mapping consistent approximations into consistent approximations, where the function applied to the limit of the inputs, is the same as the limit of the corresponding outputs.

In summary, continuous functions preserve the limits of consistent information.

5. **A domain has an *effectively given basis*.**
This is the final ingredient of the framework described by Scott, and is the most technical of his requirements by far. It introduces some constraints of a topological nature ensuring that all entities are the limits of finite approximations, largely echoing the situation of the real numbers being limits of convergent rational number intervals.

   The approach taken here is relatively economical and introduces only a few more concepts. The main idea is to ensure that all values in the data type are limits of directed subsets of what are called *basis* elements.

   We firstly introduce the *way-below* binary relation to capture more tightly what approximation means for a domain, T. We say that for $x, y \in T$ that x is *way-below* y, written as $x \lll y$, whenever, for all directed subsets D of T, that if $y \sqsubseteq \bigsqcup D$ then there exists a $d \in D$ such that $x \sqsubseteq d$.

   It is worth noting that if $x \lll y$ then $x \sqsubseteq y$, as one might expect. Intuitively, what $x \lll y$ says is that x *neccessarily approximates* y because whenever y approximates $d = \bigsqcup D$ for any directed subset D of T, then there is always some value $a \in D$ such that x approximates a which, in turn, approximates d. Clearly, this must also mean that $x \sqsubseteq \bigsqcup D$ for every directed subset D whenever $y \sqsubseteq \bigsqcup D$ – which explains the terminology 'way-below'.

   We can now introduce what is meant by the *basis elements* for a domain T, also known as the *compact* or *finite* elements for T.

   An element c is a *compact element* of domain T whenever $c \lll c$ – informally, this means that c necessarily approximates every element above it. Hence, for every directed subset D of T, if $c \sqsubseteq d = \bigsqcup D$, then there exists an element $a \in D$ such that $c \sqsubseteq a$. We also define the set of all *basis elements* for T to be:

$$\begin{aligned} K(T) &= \{\ c \in T \mid c \lll c\ \} \\ &= \{\ c \in T \mid c \text{ is compact}\ \} \end{aligned}$$

Finally, putting all this together, we say that domain T is *algebraic* whenever:

a. For every $v \in T$, we have that $v = \bigsqcup\{\, c \in K(T) \mid c \sqsubseteq v \,\}$
b. The basis set $K(T)$ is at most *countably* infinite.

Technically, what is called a domain here is known as an *ω-algebraic domain*. As hinted at earlier, every element of the domain T is then the limit of all the compact (or basis) elements that approximate it.

Interlude : Domains, least fixed points and recursion

Here we take an interlude to show how domain theory explains the idea of recursion for continuous functions, through the use of least fixed points of appropriately given *functionals*. Let D be any domain and let $f : [D \to D]$ be a Scott-continuous function from D to D.

We mathematically define the *least fix-point operator* $\mathrm{FIX} : (D \to D) \to D$ in terms of domains by:

$$\mathrm{FIX}(f) = \bigsqcup\{\, f^n(\bot) \in D \mid n \in \mathbb{N} \,\}$$

We can see that, for continuous f:

$$\begin{aligned} f(\mathrm{FIX}(f)) &= f(\bigsqcup\{\, f^n(\bot) \in D \mid n \in \mathbb{N} \,\}) \\ &= \bigsqcup\{\, f(f^n(\bot)) \in D \mid n \in \mathbb{N} \,\} \\ &= \bigsqcup\{\, f^{n+1}(\bot) \in D \mid n \in \mathbb{N} \,\} \\ &= \mathrm{FIX}(f) \end{aligned}$$

This shows that every Scott-continuous function has a fixed point.

Our next illustration uses a hypothetical programming notation to explain how recursion is related to least fixed points. Consider the humble factorial function, `fact`, mapping `Nat` to `Nat`, typically defined by:

```
letrec fact(n) = if n==0 then 1 else n*fact(n-1)
```

The **letrec** indicates here that a recursive definition is being given and semantically equates to an *implicit* use of the fix-point operator, FIX. In a call-by-name, lazy language (such as Haskell, for example) the above could equally be defined like this:

```
let fact = FIX(λ fn . λ n . if n==0 then 1 else n*fn(n-1))
```

where we also have the following recursive definition:

```
FIX f = λa . f (FIX f) a
```

We shall show how this evaluates in a specific case – and to help with that, first introduce this auxiliary function, called `bodyFact` by:

```
let bodyFact fn n = if n==0 then 1 else n*fn(n-1)
```

meaning that we have:

```
fact m  =  (FIX bodyFact) m
        =  FIX (λfn . λn . bodyFact fn n) m
```

Using this machinery, we can then evaluate `fact 2` by unrolling the above definition using the properties of fix-points:

```
fact 2  =  (FIX bodyFact) 2
        =  bodyFact (FIX bodyFact) 2
        =  if 2==0 then 1 else 2*(FIX bodyFact)(2-1)
        =  2 * (FIX bodyFact)(1)
        =  2 * (bodyFact (FIX bodyFact) 1)
        =  2 * (if 1==0 then 1 else 1*(FIX bodyFact)(1-1))
        =  2 * 1 * (bodyFact (FIX bodyFact) 0)
        =  2 * 1 * (if 0==0 then 1 else 0*(FIX bodyFact)(0-1))
        =  (2 * 1 * 1)  =  2
```

This has illustrated, via an example, how fix-points can provide a mathematically sound account of recursive definitions.

A semantic model of the untyped lambda calculus

In the final section of [19], a detailed sketch is given showing how, in broad outline, the framework could be used to build up a repertoire of basic domain-building operators for constructing compound domains out of given domains. It was there shown how these domain operators would then be used to formulate *domain equations* whose least solutions (up to isomorphism) can yield other useful domains. These solution domains would typically be used to provide the mathematical structures (e.g. lists, trees, etc.) over which recursive functions could then be defined.

The elegant idea proposed by Scott and Strachey in [20] was to define the *abstract syntax* for a notation using domain equations, thereby introducing formal term structures that

embodied the syntax represented in terms of trees etc. Semantic functions may then be defined over these term structures by structural induction and recursion[8].

As a part of this discussion, Scott then indicated how to construct a domain so that a semantic model for the pure untyped lambda calculus could at last be given.

Consider the following infinite sequence of domains D_i, defined by:

$$D_{n+1} = [D_n \rightarrow D_n]$$

Starting from a given (non-trivial) domain for D_0 (e.g. a two-point domain), the required solution is D_∞, the limit object that is obtained by 'stitching together' all of these spaces D_n via appropriate embeddings[9] Finally, we have that:

$$D_\infty \cong [D_\infty \rightarrow D_\infty]$$

where, importantly, the isomorphism pair involved is itself produced as a by-product of constructing the domain D_∞. Such domains having a recursion to the left of the function arrow are known as *reflexive* domains.

Computable functions are continuous – but *not* vice-versa

What might have puzzled some readers here concerns the relationship between continuous functions and computable functions. For example, it might be assumed that continuity implies computable – but this is *false* in general (e.g. for infinite domains)!

All partial recursive functions can be shown to be Scott-continuous, as defined in Domain Theory [2, 26, 24]. However, just on cardinality grounds, there are far too many continuous functions for them all to be computable! The situation here is somewhat analogous to the cardinality relationship between the rational/algebraic numbers (countable) and the real numbers (uncountable).

Moving away from Complete Lattices to DCPOs

In [19], domains were taken to be Complete Lattices, which meant that domains would necessarily possess a top value, $\top$. The immediate advantage of using complete lattices was that there could be no question that the appropriate limits existed, thus neatly settling any unease there might have been on that score. However, from the point of view

[8] A popular alternative approach uses *Initial Algebra Semantics* to define semantic functions [12].

[9] Technically, continuous "embedding-projection pairs" need to be used as 'fitting' morphisms instead of the simpler embeddings, due to complications connected with the structure of function spaces (See Section 4.2 of [2]).

of defining semantic functions for programming notations, having a $\top$ element was also proving somewhat cumbersome, awkward and difficult to motivate.

It was then noticed that only *directed subsets* were required to have least upper bounds, instead of taking *every* subset in order for the theory to work out. The common approach adopted for Domain Theory since then generally uses Directed Complete Partial Orders (**DCPO**s) instead of Complete Lattices[10].

A partially ordered set P is a **DCPO** whenever:

1. P has a least element, $\bot \in P$, and
2. Every non-empty directed subset D of P has a least upper bound in P i.e. $\bigsqcup D \in P$.

By using **DCPO**s instead, this provided a helpful relaxation that simplifies some aspects of the theory and additionally increases the range of natural examples of domains, and hence applicability.

Categories, Domain Operators, and Domain Equations

It is perhaps worth saying that what Domain Theory provides overall is a mathematically rich repertoire (or *toolbox*) of ways of constructing mathematical spaces within which our computational constructions and evaluations can then exist. Accordingly, the central problem answered by Domain Theory concerns showing how to systematicly construct these mathematical semantics spaces in general.

Quite naturally, Category Theory then began to be seen as providing the formal underpinnings and an appropriate setting for the wider development of Domain Theory itself. Category Theory gives a suitably structured account of broad classes of mathematical spaces, based on the fundamental notion of transformations that preserve specific kinds of mathematical structure – from *morphisms* between objects, to *functors* between morphisms, and then *natural transformations* between functors.

This modern categorial form of Domain Theory was pioneered by Gordon Plotkin, who initiated, discovered and transformed so much of what is now known and understood about these rich mathematical spaces [26, 2]. A major part of this involved showing how recursive domain equations can be formulated and solved in various categorial settings (with Mike Smyth) [25]. Plotkin's contribution is of course not limited to Domain Theory but has substantially touched many areas of semantics and Computing Science,such as formulating the modern treatment of Operational Semantics, for example [27, 6].

[10] As a postgraduate student at Edinburgh back in the late 1970s/early 1980s, I learnt Domain Theory at the feet of Gordon Plotkin, using his 'Pisa' notes [26]. These notes generally took the more minimal approach of using ω-chain Complete Partial Orders, rather than the **DCPO**s mentioned here.

Domains	Elements	Ordering
One-point domain, $\mathbb{O}$	$\{\bot\}$	$\bot = \bot$
Two-point domain, $\mathbb{1}$	$\{\bot, \top\}$	$\bot \sqsubset \top$
Truth domain, $\mathbb{T}$	$\{\bot, \mathbf{tt}, \mathbf{ff}\}$	$\bot \sqsubset \mathbf{tt}, \bot \sqsubset \mathbf{ff}$
Number domain, $\mathbb{N}_\bot$	$\{\bot\} \cup \mathbb{N}$	$\bot \sqsubset n$, for all $n \in \mathbb{N}$

Some Basic (Flat) Domains

Name	Operation	Definitions
Lifting	$D_\bot$	• $\mathrm{up} : [D \to D_\bot], \mathrm{dn} : [D_\bot \to D]$ • $\mathrm{dn}(\mathrm{up}(d)) = d$, for any $d \in D$ • $\bot \sqsubset \mathrm{up}(d)$, for any $d \in D$
Coalesced Sum	$C \oplus D$	• $\mathrm{inl} : [C \to (C \oplus D)], \mathrm{inr} : [D \to (C \oplus D)]$ • $\mathrm{inl}(c) = \langle c, 1 \rangle$, for any $c \in C - \{\bot\}$ • $\mathrm{inr}(d) = \langle d, 2 \rangle$, for any $d \in D - \{\bot\}$ • $\mathrm{inl}(\bot) = \bot = \mathrm{inr}(\bot)$ • e.g. $\bot \sqsubset \mathrm{inl}(c)$, for any $c \in C - \{\bot\}$
Cartesian Product	$C \times D$	• $\mathrm{fst} : [(C \times D) \to C], \mathrm{snd} : [(C \times D) \to D]$ • $\mathrm{fst}(\langle c, d \rangle) = c, \mathrm{snd}(\langle c, d \rangle) = d$, for any $c \in C, d \in D$ • $\langle c, d \rangle \sqsubseteq \langle c', d' \rangle \iff (c \sqsubseteq c') \wedge (d \sqsubseteq d')$
Smash Product	$C \otimes D$	• As for Cartesian Product, except that pairing is separately *strict*: $\langle \bot, d \rangle = \bot = \langle c, \bot \rangle$
Function Space	$[C \to D]$	• The set of *Scott-Continuous* functions from C to D, ordered *pointwise*: $c \sqsubseteq c' \implies f(c) \sqsubseteq f(c')$
Strict Function Space	$[C \xrightarrow{\bot} D]$	• As above for Function Space, except that functions are strict: $f(\bot) = \bot$

Some Domain Operators

Two-point domain	$\mathbb{1} \cong \mathbb{O}_\bot$
Domain of truth values	$\mathbb{T} \cong \mathbb{1} \oplus \mathbb{1}$
Domain of natural numbers	$\mathbf{Nat} \cong \mathbb{1} \oplus \mathbf{Nat}$ $\cong \mathbb{N}_\bot$
Domain of finite lists	$\mathbf{NatList} \cong \mathbb{1} \oplus (\mathbf{Nat} \otimes \mathbf{NatList})$
Domain of finite trees	$\mathbf{NatTree} \cong \mathbb{1} \oplus \mathbf{Nat} \oplus (\mathbf{NatTree} \otimes \mathbf{Nat} \otimes \mathbf{NatTree})$
Domain of finite and infinite streams	$\mathbf{NatStream} \cong \mathbb{1} \oplus [\mathbb{1} \to (\mathbf{Nat} \times \mathbf{NatStream})]$

Some Domain Equations

This section completes our brief tour of the basic technical intricacies of lambda calculus and Domain Theory. More advanced topics necessarily had to be omitted, such as a discussion of Scott's universal domain construction, $\mathbb{P}\omega$, (see [21]), Scott's elegant reformulation of Domain Theory in terms of *Information Systems* [22], Abramsky's notion of Domains in Logical Form (see [1]) as well as the question of *full-abstraction* [3].

More technical detail can be found in the mathematical accounts already cited [26, 2]. Further details on lattices and partial orders may be found in the textbook by Davie and Priestley [10]. There are a number of well-known textbooks available covering elements of Domain Theory and mathematical semantics such as [29, 18, 33], also including the following online notes: [14].

Discussion

The remainder of this essay discusses a range of questions typically raised about Domain Theory, with the hope of dispelling some myths and misconceptions that have arisen over time.

Domain Theory seems strange and unusual

At first glance, Domain Theory seems rather strange mathematically. Who would have thought that considerations of topology, continuity and approximation would ever make any appearance within Computing Science itself, a subject that is highly focused upon discrete actions and systems behaviour? And yet, that was what was needed to solve the puzzle of the untyped lambda calculus!

As already discussed earlier, it was clear that untyped lambda calculus captures something deeply primitive about the nature of effective calculation. The fact that such a calculus was extremely simple and yet contained apparently paradoxical elements suggests that more complex systems of programming could contain at least similar sources of difficulty, with probably far greater complexity and variety to follow.

We have seen that the primary difficulty solved by Domain Theory was showing how to construct particular mathematical structures containing appropriate semantic denotations – providing a semantics modelling toolkit. The development of Domain Theory made it possible to confidently formulate and routinely write down mathematical equations corresponding to language definitions, allowing language designers to reason about what was meant through mathematical investigation.

A further reason that Domain Theory may seem at least unfamiliar to some is that, in a

strong sense, Domain Theory *necessarily* operates at the *meta-mathematical* level of the *logical models* used to interpret logical statements and definitions, rather than drawing inferences about objects *within* that logic.

Domain Theory and Engineering Specifications

For some practitioners, the preceeding discussion of Domain Theory may have seemed somewhat esoteric and at odds with their understanding and general experience of computing. For example, formal software specifications are typically based on mathematical formalisms such as Higher Order Logic and Typed Set Theory (e.g. Z and VDM), all of which seemingly have little or nothing to do with the arcane elements of Domain Theory.

It is true that Domain Theory does not make a direct and *explicit* appearance in those contexts. However, that being said, the study of Domain Theory forms a part of the *model-theoretic foundations* of mathematical semantics more widely and has therefore *indirectly* informed and influenced those areas. Domain Theory, together with the theory of inductive definitions, is largely responsible for providing a well-founded mathematical approach to recursion and recursive definitions.

More subtly, it was realised that, with a careful set-theoretical treatment of types, a more conventional logic-based approach might be very effectively used for software specification. In that set up, all entities would be freely assigned meanings, just as in predicate logic.

However, the main sticking point would then be making sure that the sets and functions thus assigned as denotations satisfied appropriate structural induction principles and such like, so as to accord strongly with our computational experience. This amounts to an *extra-logical* requirement that the assigned models were in some sense *minimal*[11] – which in turn brings the discussion full circle back to concerns about recursive definition of various kinds, as tackled within Domain Theory.

Does untyped lambda calculus remain controversial?

Admittedly, the untyped lambda calculus did achieve some notoriety early on because of its association with an early system of logic later shown to be inconsistent (the Curry paradox – See Chapter 17B of [13]). As mentioned earlier, there were also concerns around the potential for non-termination and circularity due to self-application. As we have seen, this made näive interpretation of lambda terms as pure functions somewhat challenging!

[11] More technically, minimality corresponds to *initial* in an appropriate sense from Category Theory.

However, untyped lambda calculus is now far better understood and its equational theory is now known to be consistent, while also accepting that some terms may not *normalise*.

What about concurrency?

Concurrency and Parallelism remains a hot topic for research in mathematical semantics and Computing Science. Within Domain Theory itself, there are semantic domains given by the theory of *powerdomains*, due to Plotkin and Smyth [26, 2], as well as *event structures*, as developed by Winskel [34], all of which have application to concurrency.

How is Domain Theory used today?

At its core, Domain Theory provides two things:

- It firmly establishes that recursion and recursive definition, when appropriately guarded, can be generally understood mathematically in terms of concepts such as approximation, limit and fix-point.

  On the one hand, there are other ways in which recursion might be given a perfectly respectable understanding such as by structural induction, by (higher order) primitive recursion or by proving a theorem showing that the particular definition has a *unique* solution. However, each of these approaches generally imposed further constraints on the *form* of the recursive definition in some manner to ensure that solutions (uniquely) exist. On the other hand, Domain Theory provides a more general and less constrained foundation for the mathematical account of computational recursive definitions.

- Domain Theory shows how to construct mathematical spaces in which recursively defined entities are then guaranteed to exist.

The success of Domain Theory as a *semantic framework* providing the foundation and underpinnings of programming language semantics has encouraged the development and exploration of other kinds of semantic frameworks, such as Type Theory and Structural Operational Semantics, albeit from a more proof-theoretical starting point. Each of these have in same way benefited from Domain Theory, if only because they make use of a similar meta-mathematical toolbox of ideas and techniques.

Conclusions

What Domain Theory tells us is something profound. For example, it shows that the untyped lambda calculus is surprisingly quite meaningful after all and is more accurately understood to be a *uni-typed* system, as opposed to a type-free system. Much the same kind of situation arises with the more free-wheeling programming/scripting languages like Javascript, Python, Perl, Lisp and Prolog, in the sense that they all do possess some form of typing, except it is rather implicit, very often having some dynamic effect at runtime.

Much of modern computing has inherited ideas and concerns that were clearly evident within Domain Theory. Prior to the emergence of Programming Language Semantics and Domain Theory, concerns in Computing Science were largely restricted to automata, syntax and parsing, and, of course, complexity theory. Although Domain Theory may not be as fashionable as it once was, it does continue to shape and influence Computing Science.

Domain Theory is all about showing that our computational ideas are grounded and that they make sense by showing how to build consistent models in which our language-based constructions can be embodied. Without Domain Theory and Programming Language Semantics, how would anyone be able to consistently judge correctness, even in principle? How could we know what it means for a compiler to be correct?

Language descriptions are necessary and are used to articulate (by reasoning) what is and what is not permitted. Without a mathematical basis of some kind, it would be difficult to argue one way or another. Although Domain Theory may not make an explicit appearance, it nonetheless has a profound effect by showing that computational definitions can be given in a precise and principled fashion and in a way that ultimately 'makes sense' mathematically.

Acknowledgements

I am very grateful to co-editor Tim Denvir and the editorial team: Jonathan Bowen, John Cooke and Margaret West, all of whom kindly gave feedback on this rather long essay – which is far longer than originally planned! I also thank Chris Tofts for his helpful insights concerning SIMULA.

References

[1] Samson Abramsky,
Domain theory in logical form, Annals of Pure and Applied Logic, 51:1–77, 1991
https://www.sciencedirect.com/science/article/pii/016...065T

[2] Samson Abramsky, Achim Jung,
Domain Theory,
Pub. in: Handbook of Logic in Computer Science, Vol 3, Clarendon Press, Oxford, 1994
https://www.cs.bham.ac.uk/~axj/pub/papers/handy1.pdf

[3] Samson Abramsky, C.-H. Luke Ong,
Full abstraction in the lazy lambda calculus, Information and Computation, 105:159–267, 1993 *https://www.sciencedirect.com/science/article/pii/S08...0448*

[4] H.P.Barendregt, *The Lambda Calculus Its Syntax and Semantics*, Studies in Logic and the foundations of mathematics, North-Holland, 1981

[5] Guy Blelloch, Robert Harper, *λ-Calculus - The Other Turing Machine*, Conference paper, Carnegie-Mellon University, July 2015
https://www.cs.cmu.edu/~rwh/papers/lctotm/cs50.pdf

[6] Luca Cardelli, Marcelo Fiore, Glynn Winskel (Ed), *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, Electronic Notes in Theoretical Computer Science 172 (2007) *https://www.sciencedirect.com/journal/electronic-...nce/vol/172*

[7] Felice Cardone, J. Roger Hindley, *History of Lambda-calculus and Combinatory Logic*, Swansea University Mathematics Department Research Report No. MRRS-05-06, 2006
https://www.researchgate.net/publication/2283868...inatory_logic

[8] B. Jack Copeland, *The Church-Turing Thesis,* The Stanford Encyclopedia of Philosophy (Summer 2020 Edition), Edward N. Zalta (ed.),
https://plato.stanford.edu/archives/sum2020/entries/church-turing/

[9] O-J Dahl, E.W.Dijkstra, C.A.R.Hoare, *Structured Programming,*
Academic Press, 1972
https://dl.acm.org/doi/pdf/10.5555/1243380

[10] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*,
Cambridge, 2002 (2nd Ed.)

[11] Dina Goldin, Peter Wegner, *The Church-Turing Thesis: Breaking the Myth*, Lecture Notes in Computer Science 3526:152-168, June 2005
https://www.researchgate.net/publication/221652812_T...ing_the_Myth

[12] J.A.Goguen, J.W.Thatcher, E.G.Wagner, E.B.Wright, *Initial Algebra Semantics and Continuous Algebras*, JACM, Vol 24, Issue 1, Jan. 1977 pp 68-95

[13] J. Roger Hindley, Jonathan Seldin, *Introduction to Combinators and λ-Calculus*, London Mathematical Society Student Texts 1, Cambridge, 1986

[14] Graham Hutton, *Introduction to Domain Theory*, 5 lectures, 1994
http://www.cs.nott.ac.uk/~pszgmh/domains.html

[15] Dexter Kozen, *Church-Rosser Made Easy*, Fundamenta Informaticae 105 1-8, DOI 10.3233/FI-2010-306 IOS Press, 2010
https://www.researchgate.net/publication/220444851_..._Made_Easy

[16] Lukasz Lachowski, *On the Complexity of the Standard Translation of Lambda Calculus into Combinatory Logic*. REPORTS ON MATHEMATICAL LOGIC 53 (2018), 19–42 doi: 10.4467/20842589RM.18.002.8835
https://www.ejournals.eu/rml/2018/Number-53/art/12285/

[17] Kristen Nygaard and Ole-Johan Dahl *The Development of the SIMULA Languages* ACM SIGPLAN Notices. Vol. 13. No. 8. August 1978
https://phobos.ramapo.edu/~ldant/datascope/simulahistory.pdf

[18] David A. Schmidt, *Denotational Semantics*, Allyn and Bacon, 1986
https://people.cs.ksu.edu/~schmidt/text/DenSem-full-book.pdf

[19] Dana Scott, *Outline of a Methematical Theory of Computation*, (1977), Kiberneticheskij Sbornik. Novaya Seriya. 14. Also: PRG-02, Monograph, Oxford University Computing Laboratory, November 1970. *https://www.cs.ox.ac.uk/files/3222/PRG02.pdf*

[20] Dana Scott, Christopher Strachey, *Towards a Mathematical Semantics for Computer Languages*, PRG-06, Monograph, Oxford University Computing Laboratory, August 1971, *https://www.cs.ox.ac.uk/files/3228/PRG06.pdf*

[21] Dana Scott, *Data Types as Lattices*, SIAM J. Computing, Vol 5, pp522-587, 1976
https://www.researchgate.net/publication/213877138_...s_Lattices

[22] Dana Scott, *Domains for Denotational Semantics*, January 1982, Conference: Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings July 12-16, 1982, DOI:10.1007/BFb0012801
https://www.researchgate.net/publication/220897586_...nal_Semantics

[23] Dana Scott, *λ calculus: Then and Now*, Turing Centennial Celebration, Princeton University, May 10-12, 2012 *https://turing100.acm.org/lambda_calculus_timeline.pdf*

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

[25] Mike Smyth and Gordon Plotkin, *The category-theoretic solution of recursive domain equations*, SIAM J. Computing, 11:761-783, 1982
https://homepages.inf.ed.ac.uk/gdp/publications/Ca...ion.pdf

[26] Gordon D. Plotkin, *Domains*, Dept. of Computer Science, University of Edinburgh, 1983
https://homepages.inf.ed.ac.uk/gdp/publications/...4.ps

[27] Gordon D.Plotkin, *A Structural Approach to Operational Semantics*, University of Aarhus, Denmark, DAIMI FN-19, September 1981
https://www.cs.cmu.edu/~crary/819-f09/Plotkin81.pdf

[28] Raymond Smullyan, *To Mock a Mockingbird And Other Logic Puzzles*, Oxford, 1985

[29] R.D.Tennent, *Semantics of Programming Languages*, Prentice-Hall, 1991

[30] John Tromp, *Binary Lambda Calculus and Combinatory Logic*, 10.1142/9789812770837 0014, 2006,
https://www.researchgate.net/publication/30815197_Bin...tory_Logic

[31] A.M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society (Series 2), 42 (1936–37): 230–265
(Also in: *The Essential Turing*, B.Jack Copeland (Ed.), Oxford, 2004)

[32] A.M. Turing, *Computability and λ-definability*, J. Symbolic Logic, vol. 2 (1937), pp. 153-163

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

[34] Glynn Winskel, *An introduction to event structures* April 2006, Lecture Notes in Computer Science Vol 354, pp364-397 DOI:10.1007/BFb0013026
https://www.researchgate.net/publication/225207108_...uctures

History of Computing Collection at Swansea University

The History of Computing Collection specialises in computing before computers, formal methods, and local histories of computing. An introduction to the Collection appeared in the last issue of *FACS FACTS* (2021-1 February 2021, pp.10-17). The Collection is located on the Singleton Campus of Swansea University; it can be visited by appointment. A small number of items from the Collection are on display in the Computational Foundry, Bay Campus, which is the home of the Computer Science Department. All enquires welcome.

From the History of Computing Collection, Swansea University:

Haskell B Curry at War

Haskell B Curry (1900–1982) is known to computing students as the namesake of the functional programming language *Haskell*. In logic he is known for his work on combinatory logic which has influenced the design of modern functional programming languages (e.g., Turner's SASL, Ken Iversons APL). Combinatory logic is built on the idea of operations for building definitions of functions called *combinators*. Moses Schönfinkel (1888–1942), at Gottingen, had introduced the idea in 1924. Curry went to Gottingen where he gained a PhD in 1929. Curry solved the problem of completeness for a set of combinators. Over decades, starting in the 1930s, Curry developed their theory that became combinatory logic. Curry was very rigorous in matters of logic, like many founders of mathematical logic in the early Twentieth Century. He was interested in 'philosophical' questions about mathematical ideas commonly taken for granted and for which radical and primitive idealisations are necessary.

In the previous issue of *FACS FACTS* (2021-1 February 2021, pp.23-25), Jonathan Bowen wrote about *Moses Schönfinkel and combinatory logic*. It was in celebration of a century old pure mathematical idea with unforeseen but significant applications. Combinatory logic is yet one more example of our debt in computer science to early and (very) abstract speculations of logicians.

Curry on Programming

Curry deserves to be better known for his theorising about programming. This belongs to his war work for the United States Army's Ballistic Research Laboratory, which he moved to in 1944. In the History of Computing Collection, we have items that bring to life computing between and during in the world wars. The image (Figure 1) shows our copy of a report on programming the ENIAC by Haskell B Curry, written together Willa A Wyatt and finished in August 1946; it was declassified in 1999.

They begin on page 6 of their report with these clear explanations:

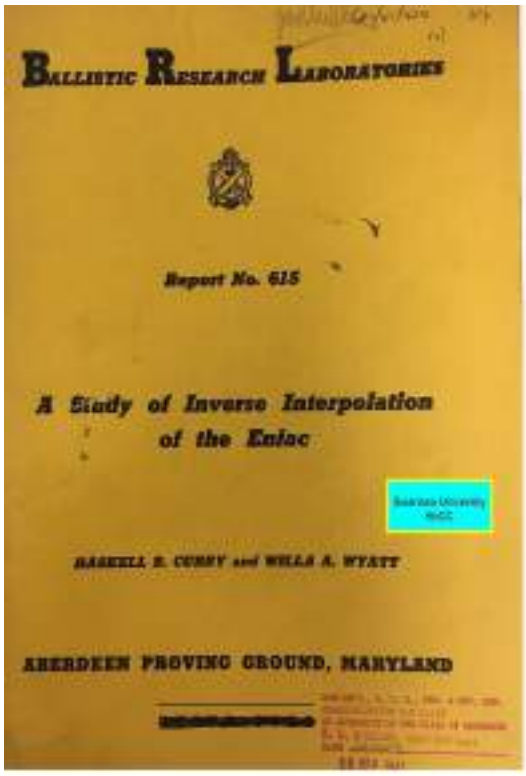


Figure 1

1.1 The problem of inverse interpolation may be stated as follows. Suppose we have a table giving values of a function x(t), and possibly some additional functions, for equally spaced values of the argument t. It is required to tabulate t and the additional quantities for equally spaced values of x.

1.2 This problem is important in the calculation of firing tables. Suppose the trajectory calculations have given us the coordinates (x,y) of the projectile as functions of t (time) and φ (angle of departure.) For the tables we want t and φ as functions of x and y; indeed we wish to determine φ so as to hit a target whose position (x,y) is known, and t is needed for the fuze setting or other purposes. This is a problem of inverse interpolation in two variables; it can be solved by two successive inverse interpolations on one variable.

Such computational work is core business for the Ballistic Research Laboratory of the Ordnance Department! The Laboratory was established in 1938 from the Research Division at the Aberdeen Proving Ground in Maryland. The production of ballistic tables for guns was a central problem for the army.

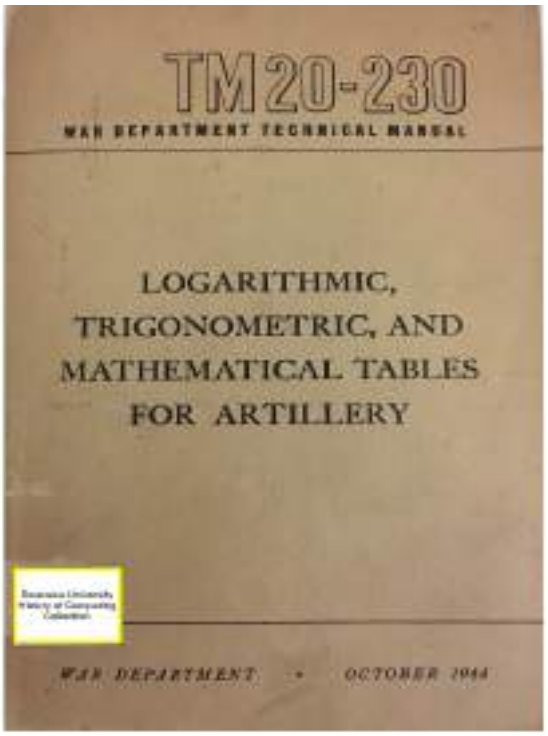


Figure 2

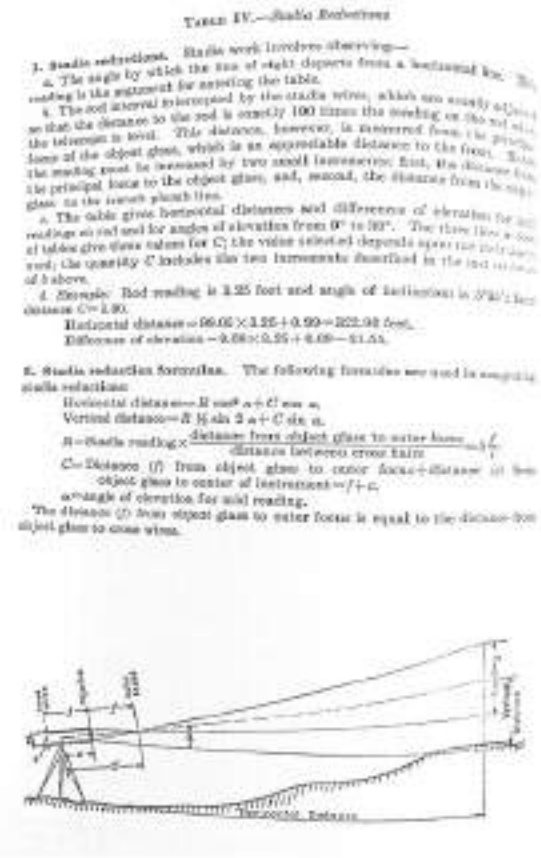
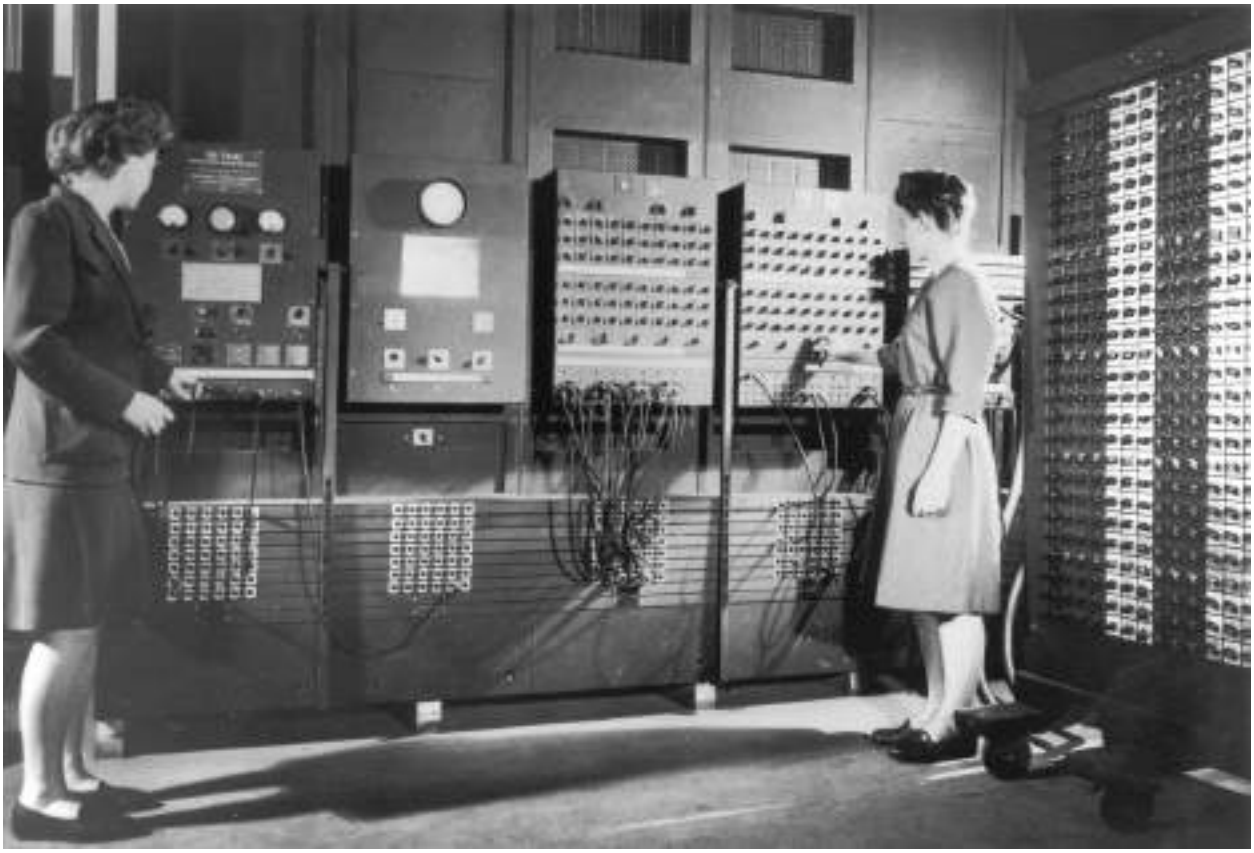


Figure 3

These were complex tables that, ideally, gave the angle of elevation required for a particular type of shell to hit a target at a given range with a given propellant. The tables gave variations to account for: atmospheric temperature; air density; wind; angle of sight; weight of projectile; muzzle velocity; and drift. Calibrating a gun was a massive computational job. See Bergin (1996).

The images show the cover and a page from our copy of a table book for gunnery calculation (Figures 2 and 3).

ENIAC



Left: Betty Jennings *Right: Frances Bilas*

Operating ENIAC (Photo: US Army).

It was the Ballistic Research Laboratory that commissioned a hugely expensive project to mechanise these tasks of table making. (Reminding us today of Babbage's first steps in mechanising table-making using the Difference Engine.) Thus, the *Electronic Numerical Integrator and Computer* – ENIAC – was designed and built in order to calculate gunnery firing tables. The project was conceived in 1943 and completed in 1945 at the University of Pennsylvania led by John Mauchly and J Presper Eckert. It was reconfigured 1947-48 and decommissioned in 1955. ENIAC was destined to be celebrated as the first programmable electronic computer in the USA.

ENIAC was indeed programmable, electronic, and a general-purpose digital computer. It was programmed by rewiring the machine: the positions of the wires and switches *was* a program! Curry's co-author Willa Wyatt was a mathematician who programmed ENIAC.

The report describes general schemes for programming inverse problems that can also be adapted to related programmes. In the 1940s and early 1950s, computer programmers were intimate with the hardware: programming was specific to individual

machines. Curry and Wyatt's concept of programming the ENIAC introduced a symbolic abstraction of the logic of the ENIAC hardware.

Curry went on to write companion reports that are noteworthy milestones for us:

- *On the composition of programs for automatic computing* (1949)
- *A program composition technique as applied to inverse interpolation* (1950)

For a fuller account of Curry's three papers on programming I recommend de Moll, Bullynck and Carle (2010). For an account of ENIAC, its history and legacy, I recommend Haigh, Priestley, and Rope (2016).

Willa Wyatt (1917-2011) was a practical ENIAC programmer. The programmers of the ENIAC were women, celebrated in Fritz (1996). She was born in in Portsmouth, New Hampshire. A graduate of New Hampshire University in 1939, she was recruited by the Moore School of Electrical Engineering at the University of Pennsylvania to work on the differential analyzer and computer sections for the ENIAC. When the ENIAC was moved to the Aberdeen Proving Grounds, Maryland, in 1946, she moved with it. There she met her husband Bill Sigmund; in 1957 they moved to Tampa, Florida where they remained: war work behind them and, as car enthusiasts, the open road ahead. Wyatt died aged 94.

References

Thomas J. Bergin (editor), *50 Years of Army Computing, From ENIAC to MSRC,* Army Research Laboratory 1996
Available at: https://www.arl.army.mil/wp-content/uploads/2019/11/ARL-SR-93-50-Years-of-Computing.pdf

Haskell B Curry and Willa A Wyatt, *A Study of Inverse Interpolation of the ENIAC*, Army Ballistic Research Laboratory, Aberdeen Proving Ground, 1946.
Available at: https://apps.dtic.mil/sti/citations/AD0640621

W. Barkley Fritz, The Women of ENIAC, *IEEE Annals of the History of Computing,* 18: 3, (1996) 13-28.

Thomas Haigh, Mark Priestley, and Crispin Rope, *ENIAC in Action: Making and Remaking the Modern Computer*, MIT Press, 2016.

L. De Mol, M. Bullynck, M. Carle, *Haskell before Haskell. Curry's contribution to programming (1946– 1950)*, In: F. Ferreira, B. Lo ̈we, E. Mayordomo, L.-M. Gomze (eds.), *Computability in Europe 2010*, Lecture Notes in Computer science, vol. 6158, Springer, 108–117. *Available at:* https://biblio.ugent.be/publication/1041602/file/6742990.pdf

John V Tucker
Department of Computer Science and History of Computing Collection
Swansea University

Domain Theory and Interaction*

Glynn Winskel

Computer & Information Sciences, Strathclyde University, Scotland

Abstract

This article traces domain theory from its beginnings as a theory of computable functions to recent connections with interactive computation. In particular, it is shown how concurrent games and strategies specialise to domain models of geometry of interaction, dialectica categories and container types.

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. In many contexts its models are the simplest we could hope for. If alone for this reason, it is here to stay. It grew up alongside the methodology of denotational semantics. The methodology of giving meaning to programming languages and systems in a compositional fashion is the only way to manage their complexity; its achievement tests our understanding and the robustness of our models.

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.

Here I'll try to explain how recent developments in understanding interactive computation in terms of concurrent strategies feed back and inform functional and domain-theoretic ways to understand computation. Several different paradigms for interaction within functional languages, logic and domain theory come about as special cases.

I have tried to be as informal as I can be 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.

*Dedicated to the memory of my father, Thomas Francis Winskel, 1925-2021.

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, Roger Penrose suggested he investigate the lambda calculus, a tool of logicians for describing and reasoning about computable functions. Meanwhile Dana Scott, at Princeton, was highly critical of the untyped nature of the lambda calculus, that it allowed such paradoxical phenomena as the application of a function to itself, something forbidden in traditional set theory. When Strachey and Scott met there were going to be fireworks, of one kind or another. At their first meeting 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. 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 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 $\mathrm{fix}(\mathrm{F})$ *constructed as the least upper bound,* $\bigsqcup_n F^n(\bot) =: \mathrm{fix}(\mathrm{F})$.

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 [1]. 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. 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 a 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. 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 ISWIM. 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.

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 Antoni Mazurkiewicz had shown how to provide semantics to jumps in imperative languages; building on earlier ideas of Egli and Milner, Plotkin had extended domain theory to a treatment of nondeterministic and parallel programs through his powerdomain—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 *denotational 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. 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 [2]. There are ways to give denotational semantics to nondeterministic dataflow but they lie outside traditional domain theory—a point we shall come back to.

From LCF, Plotkin got the idea of studying its core programming language PCF, a tradition that continues in testing new theories with some extra feature or other in the presence of function spaces [3]. 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) [4]. 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 [5]. 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 and then with his path-breaking discovery of linear logic through special kinds of dI-domains called *coherence spaces* related by stable functions.

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* [6]. 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 [7]. 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 maps of *containers* in functional programming—see Section 9.4.)

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 a view of computation based on interaction.

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 [8], 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 a structural operational semantics and the process equivalence of bisimulation [9]. Instead, Tony Hoare, with Steve Brookes and Bill Roscoe, proposed a purpose-built domain of failure sets for Communicating Sequential Processes (CSP) [10]. 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 [11, 12]. 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 [13, 14].

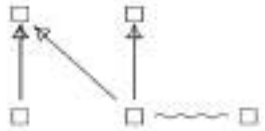
In its simplest form, an event structure *is*

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

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. The set of* configurations $\mathcal{C}(E)$ *consists of subsets of events which are left-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,



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 will be 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. A Petri net carried much more structure than could be supported by a point in a poset 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; relations between the different categories of models could be expressed as adjunctions; and helped systematise the equivalences on concurrent processes [15, 16]. This helped separate 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 fx of E' such that no two distinct events in x go to the same event in fx. 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 [17, 18]. 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—see Section 8.

6 French logic

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 about indeed denotational semantics, to which his work has nevertheless contributed enormously[1]. 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 [20]. 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 technical jargon, 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. This won't be the last time we see nontrivial structure associated with what at first sight seems a trivial degenerate case.

In studying the proofs of linear logic, Girard discovered *geometry of interaction* (GoI) [21]. 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 [22] . GoI

[1] For instance in domain models of polymorphism including System F [19], used in Section 9.4

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 [23]. Today GoI is perhaps most often viewed as an early form of game semantics—see Section 9.2, where a model of GoI emerges from a simple form of concurrent game.

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 [4] and event structures [14] 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 [24, 25, 26]. Girard's work too, often exploited the fine structure such representations.

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 [27], via Ehrhard's hypercoherences [28] or O'Hearn and Riecke's powerful logical relations [29]. 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 [30].

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 [31], while Martin Hyland, Luke Ong, and independently Hanno Nickau, discovered HO games [32]. 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.

But the 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, 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 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 [33, 34, 35, 36, 37], a considerable unification occurs with the systematic use of event structures to formalise concurrent games and strategies through their causal structure [38, 39].

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.

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 +) or Opponent (marked −). 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 A is *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.

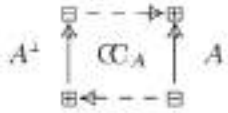
With games as event structures the history of a play of a game is no longer described by a sequence of moves, but by a partial order expressing their causal dependency. The transition from total to partial order brings in its wake technical difficulties and potential for undue complexity unless it's done artfully. Fortunately one can harness the mathematical tools developed for interacting processes, specifically on event structures [14, 15].

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 [40, 41], 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?

First, 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 $\mathbb{C}_A$ 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:



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 (nondeterministic) strategy (for Player) in which Player makes their move if Opponent makes one of theirs. It is represented by the map

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 $\boxminus \multimap \boxplus$, 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 [38].

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. That copycat is deterministic is due precisely to the game being *race-free*. (The strategy illustrated above is not deterministic.)

We shall shortly have use for two extensions to concurrent games: with winning conditions and with imperfect information.

The axioms on strategies entail a formal connection with presheaf models mentioned earlier in Section 5 and through them with Scott domains [42]. In characterising the configurations of the copycat strategy $\mathbb{C}_A$, for a game A, an important partial order on configurations appears: configurations of copycat correspond to those configurations $x \| y$ of $A^\perp \| A$ which are in the Scott order *$y \sqsubseteq_A x$, associated with undoing Opponent moves from y then executing Player moves to arrive at x. The Scott order of games extends and connects with Dana Scott's information order on domains.*

The Scott order of games is surprisingly important. In particular, a strategy in a game is a (special) presheaf over its configurations under the Scott order; essentially because the dual game reverses the Scott order, a strategy between games is a (special) profunctor—though in a way that only respects composition laxly. A profunctor directly reduces to a Scott approximable mapping so relating strategies to a domain-theoretic model.

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 [43].

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 [44]. 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.*

9 Special cases

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 $\tau \circledast \sigma$ of a deterministic strategy σ with a deterministic counterstrategy τ 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. What is perhaps surprising is that by exploiting the richer structure concurrent games we can recover

other familiar paradigms, not traditionally tied to games, or if so only somewhat informally. We start by recovering Berry's stable domain theory. The other examples, from logic and functional programming, are to do with ways of handling interaction within a functional approach.

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 polarities on events. 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.

9.1 Stable spans and stable functions

They might seem stupid as games, but let's consider games in which all the moves are Player moves.

Consider a strategy σ from from one such purely Player game A to another B, in other words a strategy in the game $A^{\perp} \parallel B$. This is a map $\sigma : S \to A^{\perp} \parallel B$ which is receptive and innocent. Notice that in $A^{\perp} \parallel B$ all the Opponent moves are in $A^{\perp}$ and all the Player moves are in B. The only new immediate causal connections, beyond those in $A^{\perp}$ and B, that can be introduced are those from what is now an Opponent move of $A^{\perp}$ to a Player move in B. Beyond the causal dependencies of the games, a strategy σ can only make a Player move in B causally depend on a finite subset of Opponent moves in $A^{\perp}$.

When σ is deterministic, all conflicts are inherited from conflicts between Opponent moves. Then, deterministic strategies correspond exactly to Berry's stable functions from the domain of configurations of A to the domain of configurations of B. Moreover this correspondence respects composition. We recover Berry's stable functions as the subcategory of deterministic strategies between concurrent games comprising purely Player moves. In the case where the games are further restricted to have trivial identity causal dependency we recover Girard's coherence spaces and their maps. (We obtain all of Berry's dI-domains when we allow slightly more general event structures than those here, which for simplicity we have based on a binary conflict; the more general event structures include Girard's *qualitative domains*.)

When σ may be nondeterministic, it corresponds to a stable span, a form of many-valued stable function which has been discovered, and rediscovered, in giving semantics to nondeterministic dataflow [45, 2, 46]. Recall from Section 4 that nondeterministic dataflow is problematic as far as traditional domain theory is concerned. Concurrent strategies support a *trace* operation which on stable spans coincides with the once-tricky feedback operation of nondeterministic dataflow [46].

If we were to extend purely Player games with winning conditions, to specify a subset of winning configurations, the stable spans and functions that would

ensue from winning strategies would send winning configurations to winning configurations.

Several important categories of domains have arisen as subcategories of concurrent games. We show that in a similar way, we obtain geometry of interaction, dialectica categories, containers, lenses, open games and learners, and optics by moving to slightly more complicated subcategories of concurrent games, sometimes with winning conditions and imperfect information.

9.2 Geometry of Interaction

Let's now consider slightly more complex games. A *GoI game* comprises a parallel composition $A := A_1 \| A_2$ of a purely-Player game A_1 with a purely-Opponent game A_2. Consider a strategy σ from a GoI game $A := A_1 \| A_2$ to a GoI game $B := B_1 \| B_2$. Rearranging the parallel compositions,

$$A^\perp \| B = A_1^\perp \| A_2^\perp \| B_1 \| B_2 \cong (A_1 \| B_2^\perp)^\perp \| (A_2^\perp \| B_1)\,.$$

So σ, as a strategy in $A^\perp \| B$, corresponds to a strategy from the purely-Player game $A_1 \| B_2^\perp$ to the purely-Player game $A_2^\perp \| B_1$. We are back to the simple situation considered in the previous section, where we considered strategies between purely-Player games.

Strategies between GoI games, from A to B correspond to stable spans from $\mathcal{C}(A_1 \| B_2^\perp)$ to $\mathcal{C}(A_2^\perp \| B_1)$, and to stable functions when deterministic. Noting that a configuration of a parallel composition of games splits into a pair of configurations,

$$\mathcal{C}(A_1 \| B_2^\perp) \cong \mathcal{C}(A_1) \times \mathcal{C}(B_2) \quad \text{and}$$
$$\mathcal{C}(A_2^\perp \| B_1) \cong \mathcal{C}(A_2) \times \mathcal{C}(B_1)\,.$$

Thus deterministic strategies from A to B correspond to stable functions

$$S = \langle f, g \rangle : \mathcal{C}(A_1) \times \mathcal{C}(B_2) \to \mathcal{C}(A_2) \times \mathcal{C}(B_1)\,,$$

associated with a pair of stable functions $f : \mathcal{C}(A_1) \times \mathcal{C}(B_2) \to \mathcal{C}(A_2)$ and $g : \mathcal{C}(A_1) \times \mathcal{C}(B_2) \to \mathcal{C}(B_1)$. Such maps are obtained by Abramsky and Jagadeesan's GoI construction, though now starting from stable domain theory [22].

The composition of deterministic strategies between GoI games, σ from A to B and τ from B to C coincides with the composition of GoI given by "tracing out" B_1 and B_2. Precisely, suppose σ corresponds to the stable function

$$S : \mathcal{C}(A_1) \times \mathcal{C}(B_2) \to \mathcal{C}(A_2) \times \mathcal{C}(B_1)$$

and τ to the stable function

$$T : \mathcal{C}(B_1) \times \mathcal{C}(C_2) \to \mathcal{C}(B_2) \times \mathcal{C}(C_1)\,.$$

Then their composition $\tau \odot \sigma$ corresponds to the stable function taking $(x, w) \in \mathcal{C}(A_1) \times \mathcal{C}(C_2)$ to $(x', w') \in \mathcal{C}(A_2) \times \mathcal{C}(C_1)$ in the least solutions to the equations

$$(x', z) = S(x, y) \quad \text{and} \quad (y, w') = T(z, w)$$

—given, as in Kahn dataflow networks, by taking a least fixed point.

It is straightforward to extend GoI games with winning conditions. The winning conditions on a GoI game $A := A_1 \| A_2$ pick out a subset of the configurations $\mathcal{C}(A)$ so amount to specifying a property $W_A(x_1, x_2)$ of pairs (x_1, x_2) in $\mathcal{C}(A_1) \times \mathcal{C}(A_2)$. That a deterministic strategy from GoI game A to GoI game B is winning amounts to

$$W_A(x, g(x, y)) \implies W_B(f(x, y), y)\,,$$

for all $x \in \mathcal{C}(A_1), y \in \mathcal{C}(B_2)$, when expressed in terms of the pair of stable functions the strategy determines. In particular, a deterministic winning strategy in an individual GoI game $B := B_1 \| B_2$ with winning conditions W_B corresponds to a stable function $f : \mathcal{C}(B_2) \to \mathcal{C}(B_1)$ such that $\forall y \in \mathcal{C}(B_2).\ W_B(f(y), y)$.

Unlike dI-domains with stable functions, with stable spans, the operation of parallel composition $\|$ is no longer a product; stable spans form a monoidal-closed and not a cartesian-closed category. For this reason, general, not just deterministic, strategies between GoI games cannot be expressed simply as lenses but instead are a form of *optics* [47].

9.3 Dialectica games

Dialectica categories were devised in the late 1980's by Valeria de Paiva in her Cambridge PhD work with Martin Hyland [48]. The motivation then was that they provided a model of linear logic underlying Kurt Gödel's *dialectica interpretation* of first-order logic. They have come to prominence again recently because of a renewed interest in their maps, in a variety of contexts from formalisations of reverse differentiation and back propagation, open games and learners, and as an early occurrence of maps as lenses.

We obtain a particular dialectica category, based on Berry's stable functions, as a full subcategory of deterministic strategies on dialectica games. Dialectica games are obtained as GoI games of imperfect information, intuitively by not allowing Player to see the moves of Opponent.

A *dialectica game* is a GoI game $A = A_1 \| A_2$ with winning conditions, and with imperfect information given as follows. The imperfect information is determined by particularly simple order of access levels: $1 \lessdot 2$. All Player moves, those in A_1, are assigned to 1 and all Opponent moves, are assigned to 2. It is helpful to think of the access levels 1 and 2 as representing two rooms separated by a one-way mirror allowing anyone in room 2 to see through to room 1. In a dialectica game, Player is in room 1 and Opponent in room 2. Whereas Opponent can see the moves of Player, and in a counterstrategy make their moves dependent on those of Player, the moves of Player are made blindly, in that they cannot depend on Opponent's moves.

Although we are mainly interested in strategies between dialectica games it is worth pausing to think about strategies in a single dialectica game $A = A_1 \| A_2$ with winning conditions W_A. Because Player moves cannot causally depend on Opponent moves, a deterministic strategy in A corresponds to a configuration

$x \in \mathcal{C}(A_1)$; that it is winning means $\forall y \in \mathcal{C}(A_2).\ W_A(x,y)$. So to have a winning strategy for the dialectica game means

$$\exists x \in \mathcal{C}(A_1) \forall y \in \mathcal{C}(A_2).\ W_A(x,y)\,.$$

(Winning strategies in general are a little more complicated to describe because they can branch nondeterministically.)

Consider now a deterministic winning strategy σ from a dialectica game $A = A_1 \| A_2$ with winning conditions W_A to another $B = B_1 \| B_2$ with winning conditions W_B. Ignoring access levels, σ is also a deterministic strategy between GoI games, so corresponds to a pair of stable functions

$$f : \mathcal{C}(A_1) \times \mathcal{C}(B_2) \to \mathcal{C}(B_1) \text{ and } g : \mathcal{C}(A_1) \times \mathcal{C}(B_2) \to \mathcal{C}(A_2)\,.$$

But moves in B_2 have access level 2, moves of B_1 access level 1; a causal dependency in the strategy σ of a move in B_1 on a move in B_2 would violate the access order $1 < 2$. That no move in B_1 can causally depend on a move in B_2 is reflected in the functional independence of f on its second argument. As a deterministic strategy between dialectica categories, σ corresponds to a pair of stable functions

$$f : \mathcal{C}(A_1) \to \mathcal{C}(B_1) \text{ and } g : \mathcal{C}(A_1) \times \mathcal{C}(B_2) \to \mathcal{C}(A_2)\,.$$

That σ is winning means

$$W_A(x, g(x,y)) \implies W_B(f(x), y)\,,$$

for all $x \in \mathcal{C}(A_1), y \in \mathcal{C}(B_2)$. Lenses f, g satisfying this winning condition are precisely the maps of de Paiva's construction of a dialectica category from Berry's stable functions.

Such pairs of functions are the *lenses* of functional programming where they were invented to make composable local changes on data-structures [49, 50]. We recover their at-first puzzling composition from the composition of strategies. Let σ be a deterministic strategy from dialectica game A to dialectica game B; and τ a deterministic strategy from B to another dialectica game C. Assume σ corresponds to a pair of stable functions f and g, as above, and analogously that τ correponds to stable functions f' and g'. Then, the composition of strategies $\tau \odot \sigma$ corresponds to the composition of lenses: with first component $f' \circ f$ and second component taking $x \in \mathcal{C}(A_1)$ and $y \in \mathcal{C}(C_2)$ to $g(x, g'(f(x), y))$.

The characterisation of nondeterministic strategies between dialectica games is more complicated to describe; they are optics based on stable spans [47].

Girard's variant

In the first half of de Paiva's thesis she concentrates on the construction of dialectica categories. In the second half she follows up on a suggestion of Girard to explore a variant. This too is easily understood in the context of concurrent games: imitate the work of this section, with GoI games extended with imperfect

information, but now with access levels modified to the discrete order on $1, 2$. Then the causal dependencies of strategies are further reduced and deterministic strategies from $A = A_1 \| A_2$ to $B = B_1 \| B_2$ correspond to pairs of stable functions

$$f : \mathcal{C}(A_1) \to \mathcal{C}(B_1) \text{ and } g : \mathcal{C}(B_2) \to \mathcal{C}(A_2)\,.$$

Combs

Through another variation we obtain the generalisation of lenses to *combs*, useful in quantum architecture and information [51, 52]. Combs arise as strategies between *comb games* which, at least formally, are an obvious generalisation of dialectica games; their name comes from their graphical representation as structures that look like (hair) combs, with each tooth representing a transformation from input to output. An *n-comb game*, for a natural number n, is an n-fold parallel composition $A_1 \| A_2 \| \cdots \| A_n$ of purely-Player or purely-Opponent games A_i of alternating polarity, so the polarities of A_i and A_{i+1} are different; it is a game of imperfect information associated with access levels $1 < 2 < \cdots < n$ with moves of component A_i having access level i. Dialectica games are 2-comb games. Discussions of causality in science, and quantum information in particular, are often concerned with what causal dependencies are feasible; then structures similar to orders of access levels are used to capture *one-way signalling*, as in dialectica games, and *non-signalling*, as in Girard's variant.

9.4 Containers

A *container game* is a game of imperfect information A w.r.t. access levels $1 < 2$; each Player move of A is sent to 1 and each Opponent move to 2. Thus within A there can be all manner of causal dependencies but *never* of the form $\boxminus \leq \boxplus$ in which a Player move causally depends on an Opponent move.

The configurations of a container game A have a dependent-type structure. Opponent moves can causally depend on Player moves, but not conversely. Let A_1 denote the subgame comprising the substructure of A consisting of purely-Player moves. A configuration $x \in \mathcal{C}(A_1)$ determines a subgame A_2/x comprising the substructure of A consisting of all those Opponent moves for which all the Player moves on which they depend appear in x. A configuration of A breaks down uniquely into a union $x \cup y$ where $x \in \mathcal{C}(A_1)$ and $y \in \mathcal{C}(A_2/x)$. We can see the configurations of a container game A as forming a dependent-sum type $\Sigma_{x \in \mathcal{C}(A_1)} A_2/x$.[2] In this way a container game corresponds to a container type, familiar from functional programming [54].

We can of course extend a container game A with winning conditions which we identify with a property W_A of the dependent-sum type $\Sigma_{x \in \mathcal{C}(A_1)} A_2/x$. A deterministic winning strategy in the container game corresponds to a configuration $x \in \mathcal{C}(A_1)$ such that $\forall y \in \mathcal{C}(A_2/x).\ W_A(x, y)$.

[2] In this section we rely on the fact that dI-domains and stable functions support the construction of dependent types—shown by Coquand, Gunter and the author, following ideas of Girard [53].

A deterministic strategy σ from a container game A to a container game B can be shown to correspond to a map of container types: a pair of stable functions

$$f : \mathcal{C}(A_1) \to \mathcal{C}(B_1) \quad \text{and} \quad g : \Pi_{x \in \mathcal{C}(A_1)}[\mathcal{C}(B_2/f(x)) \to \mathcal{C}(A_2/x)],$$

where we have relied on the fact that di-domains and stable functions support function spaces and dependent product types [53]. For container games A and B, with winning conditions W_A and W_B respectively, the strategy from A to B would be winning if, and only if,

$$W_A(x, g_x(y)) \implies W_B(f(x), y),$$

for all $x \in \mathcal{C}(A_1)$, $y \in \mathcal{C}(B_2/f(x))$. The correspondence respects composition. Container types built on dI-domains and stable functions arise as a full subcategory of concurrent games.

What about general, possibly nondeterministic, strategies between container games? Such strategies are a form of optics, extended to container types.

We won't treat symmetry in concurrent games at all here, but extending games with symmetry is important in many applications; with the addition of symmetry to a game configurations form a nontrivial category, not merely a partial order based on inclusion [55].

10 Enrichment

Concurrent strategies have been extended with probability, also with continuous probability distributions, functions on real numbers, as well as quantum structure.[3]

The enrichments specialise to the cases considered in the last section. For example, probabilistic strategies specialise to optics based on Markov kernels when between dialectica games. So work on concurrent strategies can transfer to situations of interest in functional programming and domain theory, and their extensions into probabilistic and quantum programming. This is also relevant to enrichments of open games and learners [56, 57], which can be viewed as parameterised dialectica categories. Of course there are lots of questions. How do enrichments from concurrent strategies compare with existing attempts? And when they don't already exist, how can the specialisations be characterised and simplified?

One area neglected 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 [58, 59]. I don't presently understand to what extent enrichment of concurrent strategies relates to effects.

[3] All fall within a general way to support enrichment of concurrent strategies, based on ideas developed with Marc de Visme and Pierre Clairambault.

11 Conclusion

I believe this article demonstrates the lasting power of domain theory. 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 simplify concurrent strategies. Only a fool would use a complicated model when a simpler one is available! In many contexts domain theory provides the simplest models we know.

Acknowledgments

A big thank you to Jonathan Bowen and Brian Monahan for inviting the article, to Brian and Tim Denvir for help with its preparation. I'm grateful for discussions with Fredrik Nordvall Forsberg, Bruno Gavranovic, Neil Ghani, Dan Ghica, Samuel Ben Hamou, Jules Hedges, and Jérémy Ledent. Samuel Ben Hamou, ENS Paris Saclay, tackled the verification of the early dialectica-games section for his student-internship.

References

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

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

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

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

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

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

[7] 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

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

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

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

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

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

[13] 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)

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

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

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

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

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

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

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

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

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

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

[24] 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

[25] 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

[26] 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

[27] 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

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

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

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

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

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

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

[34] 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

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

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

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

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

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

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

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

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

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

[44] 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)

[45] Nygaard, M.: Domain theory for concurrency. PhD Thesis, Aarhus University (2003)

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

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

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

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

[50] 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

[51] Chiribella, G., D'Ariano, G.M., Perinotti, P.: Quantum circuit architecture. Physical Review Letters **101**(6) (Aug 2008)

[52] Kissinger, A., Uijlen, S.: A categorical semantics for causal structure. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017, IEEE Computer Society (2017) 1–12

[53] Coquand, T., Gunter, C.A., Winskel, G.: Di-domains as a model of polymorphism. In Main, M.G., Melton, A., Mislove, M.W., Schmidt, D.A., eds.: Mathematical Foundations of Programming Language Semantics, 3rd Workshop, Tulane University, New Orleans, Louisiana, USA, April 8-10, 1987, Proceedings. Volume 298 of Lecture Notes in Computer Science., Springer (1987) 344–363

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

[55] Castellan, S., Clairambault, P., Winskel, G.: Symmetry in concurrent games. In: Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014, ACM (2014)

[56] Ghani, N., Hedges, J., Winschel, V., Zahn, P.: Compositional game theory. In Dawar, A., Grädel, E., eds.: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, ACM (2018) 472–481

[57] Fong, B., Spivak, D.I., Tuyéras, R.: Backprop as functor: A compositional perspective on supervised learning. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019, IEEE (2019) 1–13

[58] 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

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

Understanding Programming Languages

By Cliff Jones, Springer 2020

Reviewed by: Adrian Johnstone

Royal Holloway and New College
University of London

July 2021

Cliff's new book on programming language semantics is a distillation of the material he has taught at Newcastle for over a decade now, using VDM-like notation to develop (mostly) operational descriptions of (mostly) imperative programming languages. Anybody who knows Cliff or has heard him speak will know what style to expect here – confident, authoritative and challenging in the best way, so that good students are encouraged to pursue their own ideas within a formal framework.

A particular strength is the emphasis on the semantics of concurrency. We all know that reasoning informally about concurrency is very hard, and of course reasoning about meta-concurrency (i.e. notations for expressing concurrent programs) is a whole other level of challenge. This presents an educational opportunity. From the perspective of folk who want to increase the formal content in undergraduate courses, Cliff's advanced chapters on concurrency are particularly helpful, since many elementary books on programming language design and implementation struggle to get past toy examples. Students thus become demotivated since they perceive the entry price for formal methods as being high, but they then never get to see hard problems being solved. Cliff does a fine job of showing how formal semantics can save language designers from themselves, and in the process strengthens the case for formal methods on undergraduate courses.

Before looking in detail at the contents, I should do that full-disclosure thing. This cannot be an entirely objective review since I've known Cliff for many years, and some time ago he sent me a signed copy of the book… In addition, Cliff solicited my review comments on a draft, and was kind enough to mention me in the acknowledgements. So, I have seen the work in progress and when Jonathan Bowen asked me if I would review it, my inclination was to demure through lack of independence. On the other hand, I teach semantics at undergraduate level too (though in a very different style) and I want to see more mainstream courses and publicise the usefulness of this new book. As a result, here is an unsurprisingly positive review that you might want to leaven with an independent reading.

The first thing to say to this BCS-FACS audience is that this is not a book about VDM. Notationally, Cliff uses essentially a subset of VDM but since much of that comprises standard mathematical notation for sets, first order predicate logic and partial functions, supplemented with familiar square bracket notation for lists, sets of bindings for maps and conventional notation for function signatures, I think that most undergraduate finalists will be able to read these clauses easily. The notation is introduced early on in digestible lumps, and there is an appendix that enumerates all of the forms used in the book, with very useful pictorial representations of their signatures. Thus the text stands alone and is self-contained.

Cliff's first three chapters are scene-setters covering (1) the need for small meta-languages with which we can reason about programming languages, (2) the specification of and translation from concrete to abstract syntax and (3) operational semantics, and in particular Structural Operational Semantics. This marks the real shift in Cliff's pedagogic approach: whereas his teaching at Manchester was based on VDM and denotational semantics, this book is firmly SOS-oriented.

Cliff then looks at the core requirements of a sequential imperative language, with chapters on: (4) types, data and representations of the store; (5) block structure, control flow and procedures with parameter passing; and (6) objects, records, the heap and functions including higher order functions.

This material is completed with a chapter on other forms of semantic description: specifically denotational and axiomatic semantics. Cliff writes eloquently of the distinction between *model oriented* styles (including SOS) in which machine state is explicitly modelled, and *property oriented* approaches in which the semantics is defined using properties of the program text.

He goes on to give a summary of the development of denotational ideas, describes the difficulties that arise when procedures are passed to procedures, and their resolution through Scott's development of domain theory, wrapping up with a sequence of pointers into the historical literature of denotational semantics.

Axiomatic semantics, refinement calculus and VDM as an aid to formal software development and verification are discussed at some length, with a link to programming language semantics. As might be expected, Cliff gives useful insights into the history of these ideas that will motivate students' reading.

For me, the most useful material is in the final three main chapters which look at (8) shared variable concurrency, (9) concurrent object orientated languages and (10) exceptions and continuations. Our students have grown up with languages that offer pragmatic support for all of these features, yet early books on semantics rarely provide ideas on how to model them formally.

A summary of the challenges presented by interacting parallel threads of computation, and of several abstractions that have been developed to allow programmers to manage

concurrency, is followed by an expansion of the SOS idea to cover nondeterministic small-step rules which are used in the rest of the material to model concurrent programming language structures. Recent research on Rely/Guarantee reasoning and Concurrent Separation Logic is then described using these new tools.

The set piece of this book is an object-oriented language (COOL) that supports concurrency via method calls, and which avoids data races by only allowing at most one method per instance to be executing at a time. The motivation for the language, and the techniques used to formally specify the constructs are laid out in detail, and an appendix summarises the full semantics. The language would need some fleshing out to become general purpose: the intention is to illustrate core concurrency issues in a few pages.

The section on exceptions and continuations mostly focusses on approaches to generalising, and capturing formally, unstructured control flow. In just a few pages it is hard to give more than an overview and pointers into the literature, which good students will use as a springboard. Perhaps not surprisingly, Cliff labels this material *optional.*

I'll conclude with some remarks on the pedagogic style, and the place that such a book would find in the curriculum context I work in. A feature of the book is the identification of some 46 *Language issues* and eight *Challenges.* These are sprinkled throughout the text in the form of boxed asides, and each makes an excellent talking point for an interactive style of teaching. Each could be expanded into a reading or implementation assignment, and together they present a seasoned overview of the many facets of programming language design. All those who have written a few examples in some putative new programming language and then moved straight to a first implementation before getting lost in the swamp of complexity would do well to have figured out their approach to these language issues on paper before they started writing code…

Throughout, the treatment is brisk and I think that my weaker students would need some further reading. In particular, compared to 40 years ago our students often grow up in a coding monoculture where median students never really advance beyond their first language. A second-year course on informal comparative programming languages, or a thorough reading of (say) David Watt's *Programming Language Design Concepts* would be helpful in expanding students' consciousness away from Java. Another area where supplementary reading might be needed is in developing some design-level facility with SOS. Cliff presents many elegant examples, but the conciseness of the text does not allow much exploration of alternative formulations of the same core formal idea. A cookbook of ideas that covers a broad spectrum of language features might be a useful adjunct – I like Hans Huttel's *Transitions and Trees.*

One of the great strengths of this book, though, is the continual reference to the research culture, both contemporary and historical. There are voluminous references, copious footnotes and a light-touch set of asides on the history of ideas in this field which I think will strongly motivate the best students to read and fully engage with the topic, and to understand that there are open questions and opportunities to make their own future contributions.

Dimensionally correct by construction: Type systems for programs

Fredrik Nordvall Forsberg and Conor McBride
Mathematically Structured Programming Group
University of Strathclyde

Webinar presented: 15/06/2021
https://www.youtube.com/watch?v=DVDvIoz9vE0

Reported by: Keith Lines, NPL

Introduction

The last FACS talk before a summer break was a webinar presented by Fredrik Nordvall Forsberg and Conor McBride of the Mathematically Structured Programming Group at the University of Strathclyde. Fredrik and Conor are also joint appointees with the National Physical Laboratory, working on a project concerned with increasing the trustworthiness of software used in measurement systems.

The webinar introduced the concept of dependent types. It explained how dependent types can be used to define versions of linear algebra operations, such as matrix multiplication, that check the dimensions [1] of measured quantities as well as calculate numeric values. Further details are provided in [2].

Fredrik and Conor at the whiteboard.

Instead of the usual slides, Fredrik and Conor provided a whiteboard-based double-act and included a physics demonstration. This approach was very entertaining and much appreciated.

Summary

A simple example, using Haskell, demonstrated how the type checking most programmers are familiar with often does not help with writing trustworthy software. Treating all elements of a type equally, when type checking, is an approximation that can have serious consequences. Dependent types increase trustworthiness by including contextual information, e.g. whether a number is being used as the index of an element in an array.

A type definition for lists demonstrated how much type and proof checkers can leave the programmer to implement and prove. E.g. list appending is an associative operation, but such basic properties are not always "built in".

A matrix where rows represent students, columns represent tests results and cells contain exam results provided a further example of context. Each cell contains information specific to a particular student and a particular exam.

Attention then turned to physics, in particular dimensional analysis. Mass (M), length (L) and time (T) are amongst these dimensions. A practical demonstration showed how dimensions can be used derive an expression for calculating the period of a pendulum. Dimensions helped derive the expression in a way analogous to how types help with writing software. The next stage was to bring these two concepts together.

As Andrew Kennedy noted in the 1990s there are strong similarities between dimensions in physics and types in programming languages. Fredrik and Conor's research presented in this section of the webinar builds on Kennedy's [3] and George Hart's [4] work.

A free abelian group on the set of fundamental dimensions, contains the dimensions that can be assigned to quantities. Type checkers that implement all this theory obviously relieve the programmer of having to implement and prove these concepts, making software more trustworthy.

Question and answers

Topics covered included:

- The theory outlined in this talk is not only applicable to functional languages. E.g. type theory has been introduced to PHP [5].
- The was a discussion about possible areas of applications and similar work undertaken the past.

References

1 BS EN ISO 80000-1:2013, Quantities and units

2 McBride C., Nordvall-Forsberg F., Type systems for programs respecting dimensions (Jan 2021)
Retrieved 27th June 2021 from University of Strathclyde
https://pureportal.strath.ac.uk/en/publications/type-systems-for-programs-respecting-dimensions

3 Kennedy A. J., Programming languages and dimensions (April 1996)
Retrieved 27th June 2021 from Cambridge University
https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf

4 George W. Hart, "Multidimensional Analysis: Algebras and Systems for Science and Engineering", Springer, 1995

5 Hack, a dialect of PHP created for Facebook
Retrieved 27th June 2021 from hacklang.org
https://hacklang.org/

ABZ 2021 Conference Report

Jonathan P. Bowen
Chair, BCS-FACS, June 2021

Introduction

The ABZ 2021 *8th International Conference on Rigorous State Based Methods* was held entirely virtually during June 2021. This incorporated presentations of papers from the planned ABZ 2020 conference, postponed due to the COVID-19 pandemic. It was held during 9–11 June 2021, preceded by a *Colloquium on the Occasion of Egon Börger's 75th Birthday* with an associated Festschrift volume, on 7 June 2021 and the *9th Rodin User and Developer Workshop* with other sessions on 8 June 2021, making five days of related online presentations in all. Thus, there were three associated Springer LNCS (Lecture Notes in Computer Science) volumes for the Festschrift (Raschke et al., 2021), ABZ 2020 (Raschke et al., 2020), and ABZ 2021 (Raschke & Méry, 2021). The events were organized by the University of Ulm in Germany, using Zoom for sessions and *wonder.me* for breaks, allowing online networking. There was no registration fee due to sponsorship.

Old school meets new school with a whiteboard presentation on Zoom, presenting work by Jean-Raymond Abrial and Dominique Cansell during the Rodin Workshop.

Colloquium on the Occasion of Egon Börger's 75th Birthday

On 7 June 2021, a one-day celebration of Egon Börger's 75th birthday with presentations and an associated Festschrift volume (Raschke et al., 2021) was held on Zoom. The event was organized through the University of Ulm in Germany by Alexander Raschke, Elvinia Riccobene, and Klaus-Dieter Schewe. A previous

Festschrift celebration for Egon Börger's 60th birthday was held in 2006 at Schloss Dagstuhl in Germany (Abrial & Glässer, 2009). Egon Börger has been the leading promulgator of the ASM (Abstract State Machines) formal method for much of his career. He also has an interesting background in his academic advisor tree, leading back to Hegel, Kant, and Leibniz, among others (Bowen, 2021).

Introduction to Egon Börger's 75th Festschrift celebration by Alexander Raschke.

Discussions on Zoom during Egon Börger's 75th Festschrift celebration.

Using `wonder.me' during breaks.

Egon Börger speaking during his 75th Festschrift celebration.

ABZ 8th International Conference on Rigorous State Based Methods

The ABZ conference series was initiated in London during 2008, when the previous ZB conference series on the Z notation and B-Method combined with the previous ASM workshops to form "ABZ" (Börger et al., 2008). The series has gradually expanded in its scope to include further state-based formal approaches such as Alloy, TLA, and VDM. It is now open to any state-based formal (or "rigorous") method.

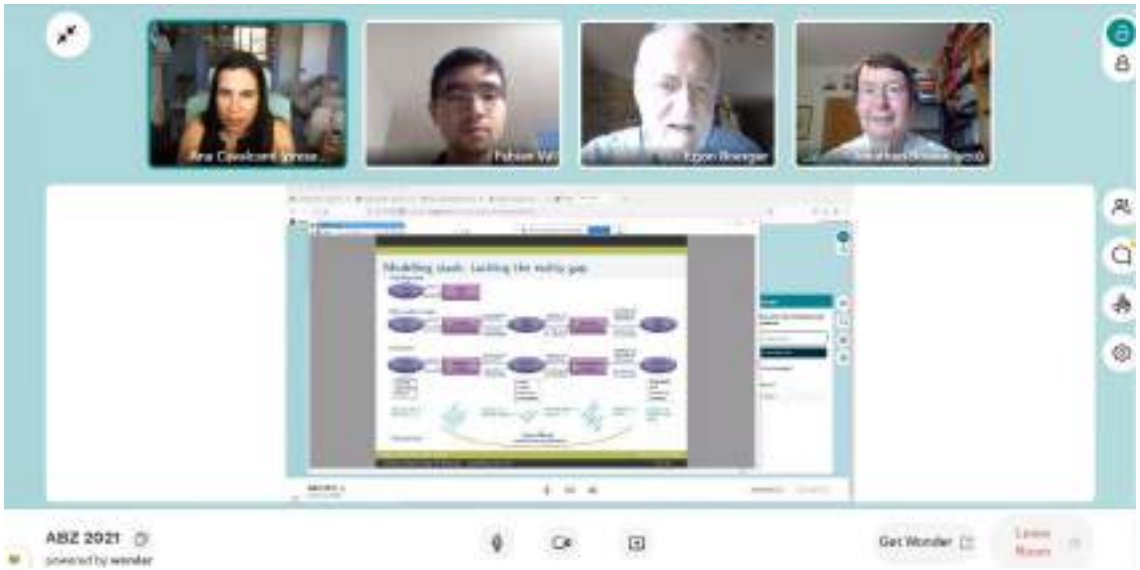
During 9–11 June 2021, papers submitted for both the ABZ 2020 and ABZ 2021 conferences were presented, in short 15-minute and longer 30-minute formats on Zoom. There were two keynote talks, each an hour in length, by Ana Cavalcanti

(Chair of FME, Formal Methods Europe) and Gilles Dowek of INRIA in France. It was originally hoped to hold the first of these conferences in 2020 but due to the Covid-19 pandemic this was delayed and eventually held completely online in combination with ABZ 2021. The ABZ 2020 proceedings was edited by Alexander Raschke, Dominique Méry, and Frank Houdek (Raschke et al., 2020) and the ABZ 2021 proceedings was edited by the first two of these editors (Raschke et al., 2021).

It is interesting to note the occurrence of various formal methods and tools in the titles of papers in the two ABZ proceedings. Event-B tops the list with 14 papers. 11 papers mention the Rodin tool, providing Event-B tool support. Next there are nine papers with ASM in the title (including two mentioning the ASMETA toolset). Alloy is mentioned in three paper titles, as is the ProB tool providing tool support for B. The Atelier B, UML-B, and UPPAAL tools are each mentioned in one title. Interestingly, TLA, VDM, and Z are not mentioned in any paper titles. So, the "A" (ASM and Alloy) and "B" (mainly Event-B with the associated Rodin and ProB tools) in "ABZ" live on, especially strongly in the case of ASM and Event-B/Rodin. However, the "Z" part has essentially disappeared and perhaps can now be considered as "the rest". C'est la vie (as we say in the UK!).

A break during ABZ 2021 before Ana Cavalcanti's keynote talk.

Discussions on Zoom during ABZ 2021.



Discussions on `wonder.me' during ABZ 2021, including an impromptu presentation.

Conclusion

The events all went very well in the circumstances, although online networking is not the same as at a real conference of course. Swapping between Zoom for presentations and *wonder.me* for breaks was not ideal. In the future, perhaps Zoom will develop to include better facilities for breaks in meetings, but *wonder.me* does allow participants to easily join discussion groups in a visual way. The next ABZ 2023 conference is planned to be at Nancy, France, organized by Dominique Méry et al. at LORIA, University of Lorraine. We hope that this can be a physical conference again! Meanwhile, for further information on ABZ 2021, see: https://abz2021.uni-ulm.de

The three proceedings associated with the ABZ 2021 conference.

References

Abrial, J.-R. and Glässer, U. (eds.) (2009). Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday. Springer, Lecture Notes in Computer Science, volume 5115. DOI: 10.1007/978-3-642-11447-2

Börger, E., Butler, M., Bowen, J.P., and Boca, P. (eds.) (2008). Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16-18, 2008, Proceedings. Springer, Lecture Notes in Computer Science, volume 5238. DOI: 10.1007/978-3-540-87603-8

Bowen, J.P. (2021). Communities and Ancestors Associated with Egon Börger and ASM. In Raschke et al. (2021), pages 96-120. DOI: 10.1007/978-3-030-76020-5_6

Raschke, A. and Méry, D. (eds.) (2021). Rigorous State-Based Methods: 8th International Conference, ABZ 2021, Ulm, Germany, June 9-11, 2021, Proceedings. Springer, Lecture Notes in Computer Science, volume 12709. DOI: 10.1007/978-3-030-77543-8

Raschke, A., Méry, D., and Houdek, F. (eds.) (2020). Rigorous State-Based Methods: 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings. Springer, Lecture Notes in Computer Science, volume 12071. DOI: 10.1007/978-3-030-48077-6

Raschke, A., Riccobene, E., and Schewe, K.-D. (eds.) (2021). Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday. Springer, Lecture Notes in Computer Science, volume 12750. DOI: 10.1007/978-3-030-76020-5

Meeting Reports

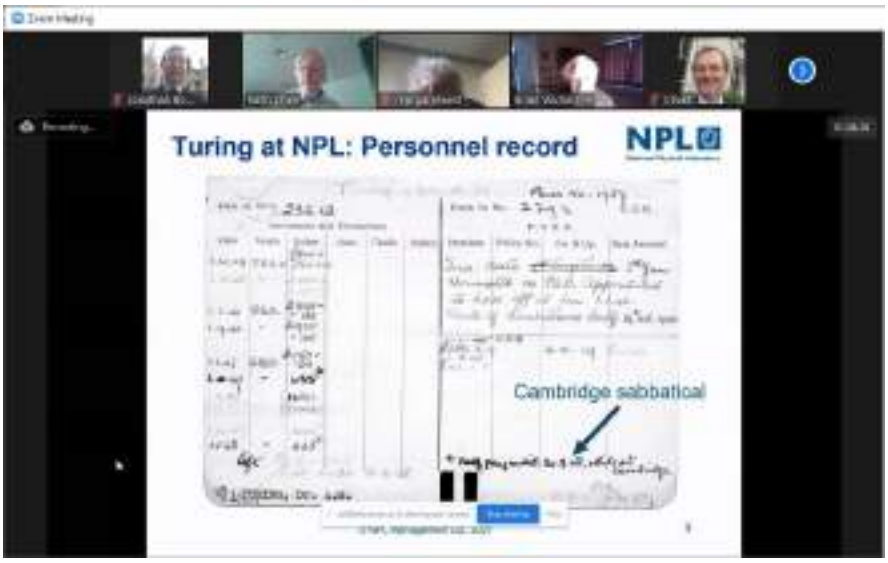
Jonathan P. Bowen
Chair, BCS-FACS, May 2021

Introduction

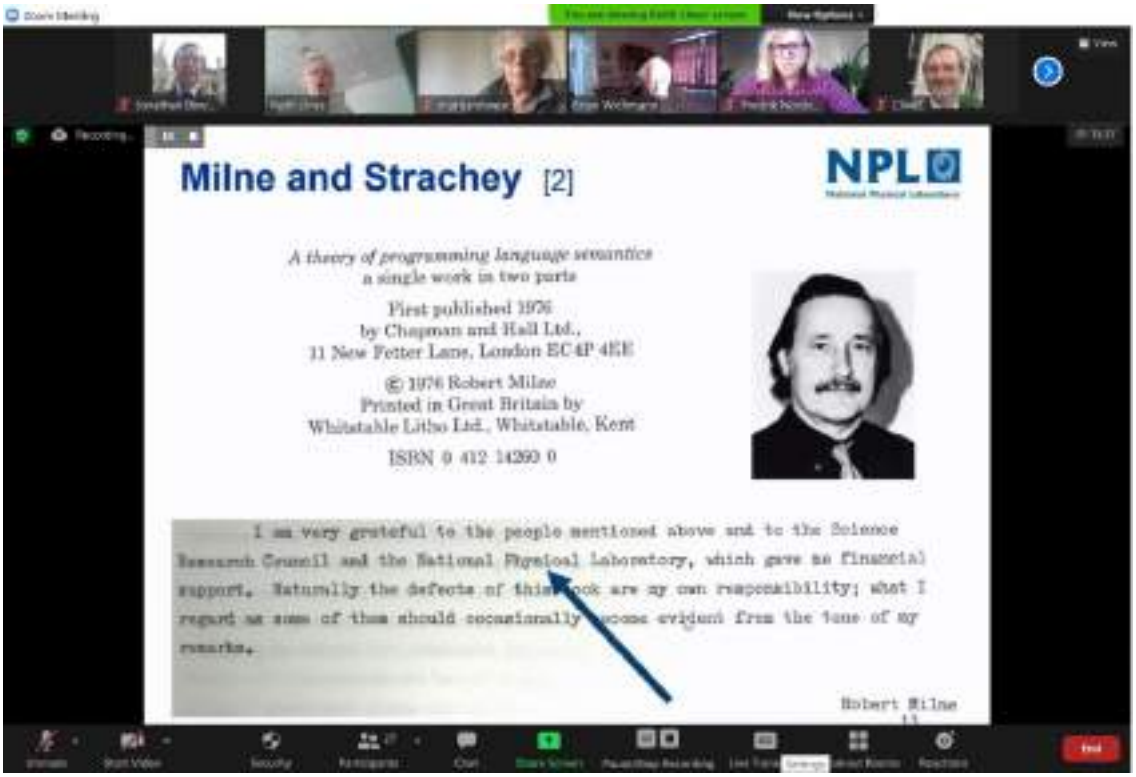
FACS has moved online for its meetings using the Zoom facilities of the BCS. This makes recording of talks easier, as well as enabling a more geographically dispersed audience. Of course, the networking opportunities are reduced, and we aim to resume meetings at the BCS London office when this is possible. It is then likely that talks will be hybrid in nature, with a real audience and an online audience, hopeful the best of both worlds. I understand that the BCS plan facilities at the BCS London office to enable this, but no timescale has been set yet.

Keith Lines, NPL

On 6 April 2021, Keith Lines, a FACS committee member based at the National Physical Laboratory (NPL), gave an interesting talk entitled, "*NPL's Experience with Formal Aspects*", covering activities at NPL in the area of formal methods. He started with a briefly introduction to perhaps NPL's most famous "formal methods" person, indeed "computer scientist", although neither terms were used with their modern meanings at the time, Alan Turing (1912–1954). The talk included Turing's hand-written NPL personnel record and NPL's connections with Robert Milne and also Christopher Strachey, a colleague of Turing and founder of the Programming Research Group at Oxford. Brian Wichmann, a retired member of NPL, was in the audience and contributed some interesting remarks during the talk. A video of the talk is available online under: https://www.bcs.org/events/2021/april/webinar-npl-s-experience-with-formal-aspects/



Alan Turing's NPL personnel record.



Robert Milne and Christopher Strachey.

Synopsis

The National Physical Laboratory's pioneering role in modern-day computing is well known; not least because of Alan Turing's design of the ACE (Automatic Computing Engine) and Donald Davies' development of packet switching. NPL has also maintained an interest in theoretical computer science and formal methods over the years. This presentation summarised NPL's work in this area, including: 1) exploring the use of formal methods in the standardisation of communications protocols; 2) a survey undertaken in the 1990s on the take up (or lack thereof) of formal methods within industry; 3) work undertaken with the Department of Computer Science of the University of York as part of the EU-funded Traceability for Computationally Intensive Metrology (TraCIM) project. NPL continues formal aspects work through joint appointments with universities. The presentation ended with a very brief overview of a project, undertaken with the University of Strathclyde, on physical dimensions and types, the subject of a FACS presentation from Strathclyde in June 2021.

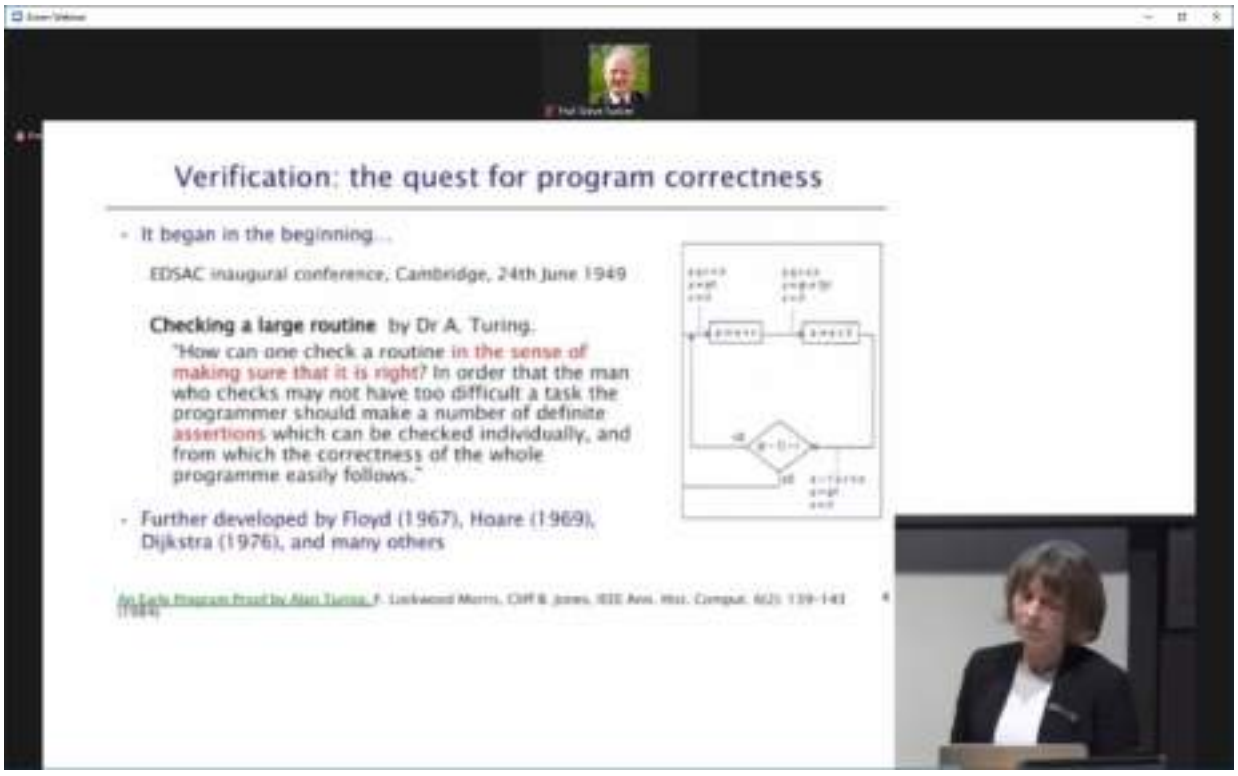
Biography

Keith Lines applies experience gained in over 30 years of working with NPL's scientists, administrators, and support staff to help ensure that NPL activities in software development continue to meet the requirements of NPL's ISO 9001 and TickITplus certifications. Formal aspects of computing have been an interest since he was a student at the University of Kent in the mid-1980s. He is a member of the BCS.

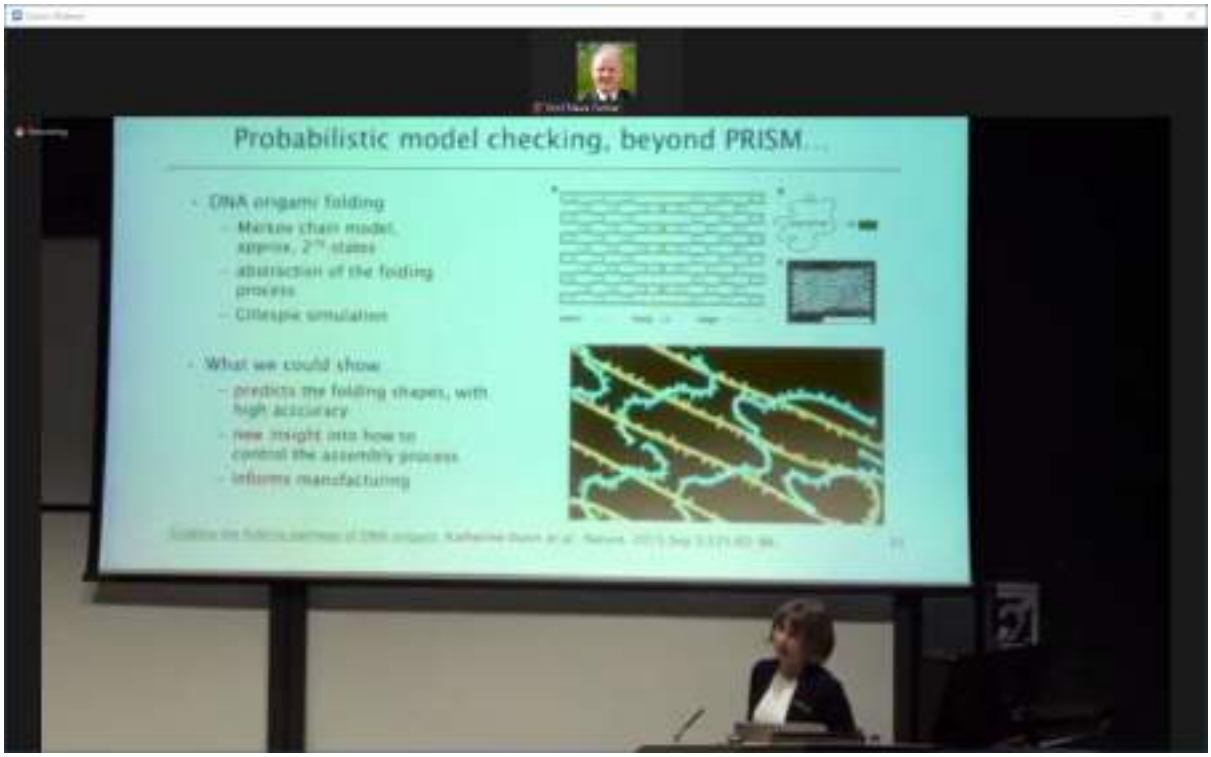
Marta Kwiatkowska, Oxford

Although not a FACS presentation, Marta Kwiatkowska of Oxford University delivered the BCS Lovelace Lecture, entitled "*Probabilistic Model Checking for the Data-rich World*", on 5 May 2021. Marta has given previous lectures to FACS and was elected a Fellow of the Royal Society (FRS) in 2019. The talk was introduced by Professor Dame Muffy Calder DBE OBE, Professor of Formal Methods, Head of the College of Science and Engineering, and Vice Principal of the University of Glasgow, as well as being another former FACS speaker. Professor Steve Furber CBE FRS of the University of Manchester chaired the talk, which concentrated on the PRISM probabilistic model checker, developed by Marta and her group at Oxford. An impressive range of applications was presented. A vote of thanks was given at the end of the talk by Professor Tony Cohn of the University of Leeds. Information on the talk is available online under:

https://www.bcs.org/events/2021/may/bcs-lovelace-lecture-202021-prof-marta-kwiatkowska/



Checking a large routine, by Alan Turing.



Probabilistic model checking beyond PRISM.

Synopsis

Computing systems have become indispensable in our society, supporting us in almost all tasks, from social interactions and online banking to robotic assistants and implantable medical devices. Since software faults in such systems can have disastrous consequences, methods based on mathematical logic, such as proof assistants or model checking, have been developed to ensure their correctness. However, many computing systems employ probability, for example as a randomisation technique in distributed protocols, or to quantify uncertainty in the environment for AI and robotics applications. Systems with machine learning components that make decisions based on observed data also have a natural, Bayesian probabilistic interpretation. In such cases, logic no longer suffices, and we must reason with probability. Probabilistic model checking techniques aim to verify the correctness of probabilistic models against quantitative properties, such as the probability or expectation of a critical event. Exemplified through the software tool PRISM (www.prismmodelchecker.org), they have been successfully applied in a variety of domains, finding and fixing flaws in real-world systems. As today's computing systems evolve to increasingly rely on automated, strategic decisions learnt from rich sources of data, probabilistic model checking has the potential to provide probabilistic robustness guarantees for machine learning. Using illustrative examples from mobile communications, robotics, security, autonomous driving and affective computing, this lecture gave an overview of recent progress in probabilistic model checking, and highlighted challenges and opportunities for the future.

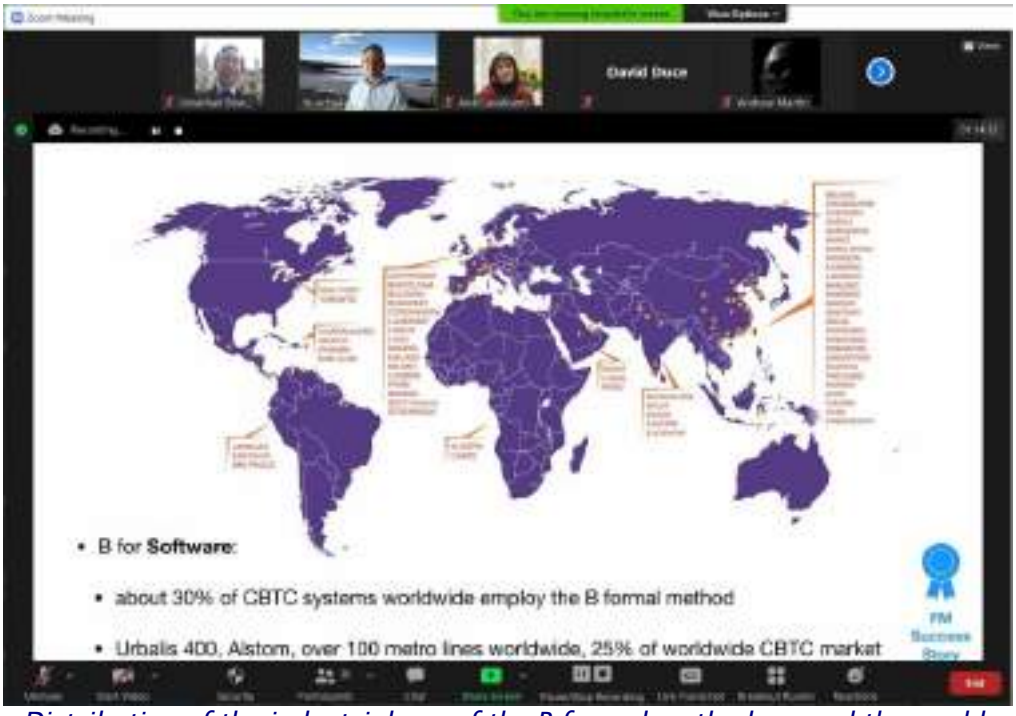
Biography

Marta Kwiatkowska is Professor of Computing Systems and Fellow of Trinity College, University of Oxford. She is known for fundamental contributions to the theory and practice of model checking for probabilistic systems. She led the development of the PRISM model checker, the leading software tool in the area. Probabilistic model checking has been adopted in diverse fields, including distributed computing, wireless networks, security, robotics, healthcare, systems biology, DNA computing and nanotechnology, with genuine flaws found and corrected in real-world protocols. Marta Kwiatkowska was awarded two ERC Advanced Grants, VERIWARE and FUN2MODEL, and is a coinvestigator of the EPSRC Programme Grant on Mobile Autonomy. She was honoured with the Royal Society Milner Award in 2018 and the Lovelace Medal in 2019, and is a Fellow of the Royal Society, ACM and BCS, and Member of Academia Europea.

Michael Leuschel, Düsseldorf, Germany

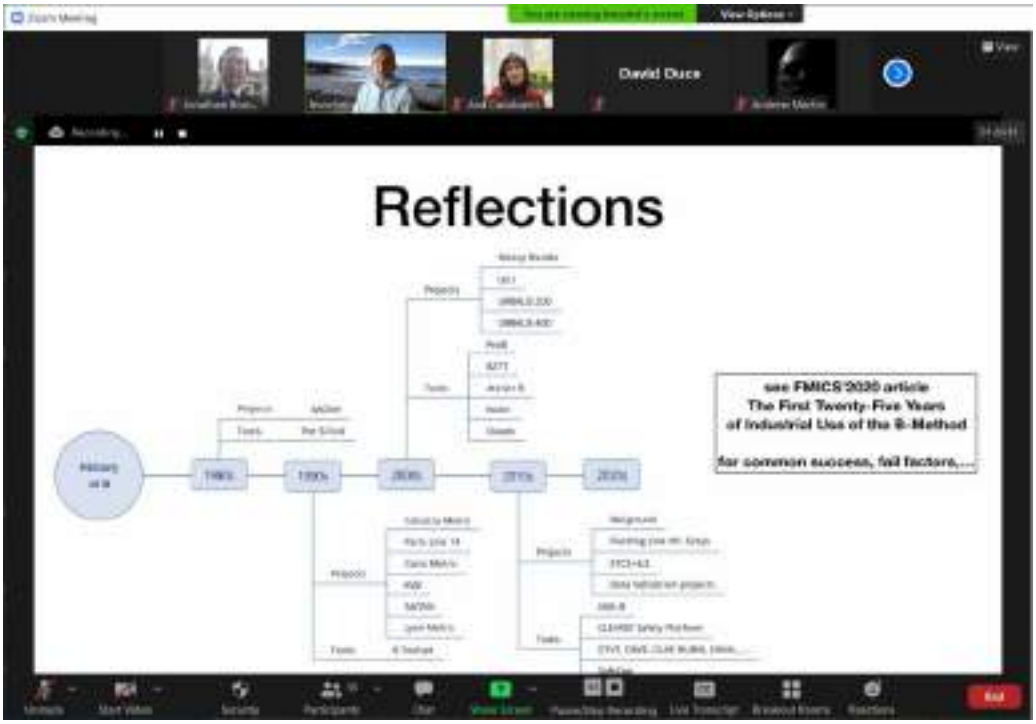
On 6 May 2021, Michael Leuschel, a professor at the Institut für Informatik of Heinrich-Heine-Universität Düsseldorf in Germany, delivered a joint Formal Methods Europe (FME) and FACS talk entitled "*New Ways of Using Formal Models in Industry*". The talk covered Michael's extensive experience of liaising with industry in the use of formal methods, mainly using the B-Method and especially Event-B, including the use of the ProB animator and model checker tool developed by Michael and his colleagues. Ana Cavalcanti, chair of FME, attended the talk, giving a brief introduction and welcome. A video of the talk is available online under:

https://www.bcs.org/events/2021/may/webinar-evening-seminar-facs-sg/



Distribution of the industrial use of the B formal method around the world.

The 1st Conference on the B method.



A graphical history of the industrial use of the B-Method.

ProB tool demonstration.

Synopsis

Advances in formal methods tools have enabled a wide variety of new ways of using formal models and for increasing the added value of formal modelling. This talk presented experience in using the B formal method for systems modelling and data validation in the railway sector. The talk started out by situating the B-Method within the realm of formal methods and providing a brief overview of twenty-five years of industrial usage. The talk then discussed various lessons learnt during the speaker's experience with formal methods, in particular for the new hybrid-level 3 European train control system specification. It discussed how to combine the various verification and validation aspects, from proof to visualization, leading to new applications such as executable prototypes or interactive requirements documents.

Biography

Michael Leuschel is full professor at the Institut für Informatik of Heinrich-Heine-Universität Düsseldorf, Germany, where he leads the Software Engineering and Programming Languages group. His research focusses on model-based problem solving using symbolic model checking. He has been one of the main developers of ProB, a successful animator, constraint solver and model checker for the B-Method. ProB is certified T2 SIL4 according to the Cenelec EN 50128 standard. Michael's research is also behind the development of the ECCE system for partial deduction.

Conclusion

We plan to continue to make FACS talks available on Zoom and then as videos via YouTube after the talk if the speaker agrees. However, we also look forward to live talks again when this is possible and aim to deliver these in hybrid mode, both at the BCS London office and online. We are looking for a volunteer to join the FACS committee and help in organizing FACS talks. This is an excellent opportunity to enable talks by people you wish to hear. If you would like to take on this role, or suggest a speaker and co-organize a single talk, please contact the Chair of FACS, Jonathan Bowen, on: *jonathan.bowen@lsbu.ac.uk*

Forthcoming events

Events Venue (unless otherwise specified):

BCS, The Chartered Institute for IT
Ground Floor, 25 Copthall Avenue, London, EC2R 7BP

The nearest tube station is Moorgate, but Bank and Liverpool Street are within walking distance as well.

23 September, 5:15pm - 8:00pm	**Webinar: Matrices of Sets** *An introduction to Matrices of Sets* **Speaker:** Renaud Di Francesco, Sony Europe BV **Synopsis:** An introduction to Matrices of Sets, i.e. tables where the position at line i and column j is occupied by a set M(i,j), instead of a number. https://www.bcs.org/events/2021/september/webinar-matrices-of-sets-bcs-formal-aspects-of-computing-science-sg-facs/

Details of all forthcoming events can be found online here:

https://www.bcs.org/membership/member-communities/facs-formal-aspects-of-computing-science-group/

Please revisit this site for updates as and when further events are confirmed.

FACS Committee

Formal Aspects of Computing Science Specialist Group

Jonathan Bowen
FACS Chair and
BCS Liaison

John Cooke
FACS Treasurer and
Publications

Roger Carsley
Minutes Secretary

Rob Hierons
LMS Liaison

Ana Cavalcanti
FME Liaison

Brijesh Dongol
Refinement
Workshop Liaison

Keith Lines
Government and
Standards Liaison

Margaret West
Inclusion Officer and
BCS Women Liaison

Tim Denvir
Co-Editor, FACS
FACTS

Brian Monahan
Co-Editor, FACS
FACTS

FACS is always interested to hear from its members and keen to recruit additional helpers. Presently we have vacancies for officers to help with fund raising, to liaise with other specialist groups such as the Requirements Engineering group and the European Association for Theoretical Computer Science (EATCS), and to maintain the FACS website. If you are able to help, please contact the FACS Chair, Professor Jonathan Bowen at the contact points below:

BCS-FACS
c/o Professor Jonathan Bowen (Chair)
London South Bank University
Email: jonathan.bowen@lsbu.ac.uk
Web: www.bcs-facs.org

You can also contact the other Committee members via this email address.

Mailing Lists

As well as the official BCS-FACS Specialist Group mailing list run by the BCS for FACS members, there are also two wider mailing lists on the Formal Aspects of Computer Science run by JISCmail.

The main list <facs@jiscmail.ac.uk> can be used for relevant messages by any subscribers. An archive of messages is accessible under:

http://www.jiscmail.ac.uk/lists/facs.html

including facilities for subscribing and unsubscribing.

The additional <facs-event@jiscmail.ac.uk> list is specifically for announcement of relevant events.

Similarly, an archive of announcements is accessible under:

http://www.jiscmail.ac.uk/lists/facs-events.html

including facilities for subscribing and unsubscribing.

BCS-FACS announcements are normally sent to these lists as appropriate, as well as the official BCS-FACS mailing list, to which BCS members can subscribe by officially joining FACS after logging onto the BCS website.