Strachey 100

Celebrating the life and research of Christopher Strachey

University of Oxford
Department of Computer Science

18th–19th November 2016

Wolfson Building, Parks Road
Oxford
OX1 3QD
United Kingdom
http://www.cs.ox.ac.uk

Organisers:

Samson Abramsky (samson.abramsky@cs.ox.ac.uk)
Troy Astarte (t.astarte@newcastle.ac.uk)
Karen Barnes (karen.barnes@cs.ox.ac.uk)
Alex Kavvos (alex.kavvos@cs.ox.ac.uk)
Cliff Jones (cliff.jones@newcastle.ac.uk)
Bernard Sufrin (bernard.sufrin@cs.ox.ac.uk)

With thanks to the Department of Computer Science at the University of Oxford, and the Strata Platform Grant, for providing funding for the event and the travel of some of those involved.

Introduction

Christopher Strachey (1916–1975) was a pioneering computer scientist and the founder of the Programming Research Group, now part of the Department of Computer Science at Oxford University. Although Strachey was keenly interested in the practical aspects of computing, it is in the theoretical side that he most indelibly left his mark, notably by creating with Dana Scott the denotational (or as he called it, 'mathematical') approach to defining the semantics of programming languages. Strachey also spent time writing complex programs and puzzles for various computers, such as a draughts playing program for the Pilot ACE in 1951. He developed some fundamental concepts of machine-independent operating systems, including an early suggestion for time-sharing, and was a prime mover in the influential CPL programming language. Strachey came from a notable family of intellectuals and artists, perhaps most famous for Christopher's uncle Lytton, a writer and member of the Bloomsbury group.

The conference will open with a morning of talks dedicated to a historical exploration of Strachey's life, chaired by Cliff Jones. The morning session will end with a panel chaired by Bernard Sufrin. After lunch, there will be a video presentation from Strachey's long-time collaborator Dana Scott, who regrets he is unable to be in the United Kingdom for this event. The afternoon session will consist of talks about the future of research inspired by Strachey at Oxford and further afield, which will be chaired by Samson Abramsky.

This booklet will provide some information on the proceedings of the conference, including a schedule of talks and short abstracts from those contributors who provided them. There is also a series of appendices with some longer papers, and a list of the items shown at an exhibition at the Bodleian Library.

Programme

Start	Name	Title
	Morning session	*(Chair: Cliff Jones)*
09:00		Opening remarks
09:15	Martin Campbell-Kelly	Strachey: the Bloomsbury years
09:45	Joe Stoy	Strachey and the Oxford Programming Group
10:15	Martin Richards	Strachey and the development of CPL
10:45		*break*
11:15	Peter Mosses	SIS, a semantics implementation system
11:45	Robert Milne	Semantic relationships: reducing the separation between practice and theory
	David Hartley	
12:15	Michael Jackson	Panel *(Chair: Bernard Sufrin)*
	Roger Penrose	
	Chris Wadsworth	
13:00		*lunch*
	Afternoon session	*(Chair: Samson Abramsky)*
14:15	Dana Scott	Video presentation
14:45	Jane Hillston	A mathematical approach to defining the semantics of modelling languages
15:15	Philip Wadler	Christopher Strachey, first-class citizen
15:45	*break*	
16:15	Hongseok Yang	Probabilistic programming
16:45	Uday Reddy	Parametric polymorphism and models of storage
17:15	Jeremy Gibbons	What are types for?
17:45		Closing remarks

Talk abstracts

Martin Richards:
'Christopher Strachey and the Development of CPL'

Christopher Strachey was the most significant contributor to the design and implementation of the programming language CPL. At the time there was little understanding of the complexities of computer language design and how type systems could cope with lists and the kinds of structures needed to represent, for instance, parse trees. The CPL project cannot be regarded as being successful since it did not result in a usable CPL compiler. The reasons being that the language became too large and complicated, there were insufficient people to implement the compiler and, in the middle of the three year project, all work had to be transferred from Edsac 2 to Titan, a newly designed version of the Ferranti Atlas computer which as yet had no operating system. Even so, we can be proud of the work that went into CPL and its influence on the design of many later languages.

Peter Mosses:
'SIS, A semantics implementation system'

During my DPhil studies, supervised by Christopher Strachey, I developed a prototype of a system for executing programs based on their denotational semantics. It involved partial evaluation of lambda-notation, implemented using Wadsworth's call-by-need algorithm. I continued the development of the system as a postdoc at Oxford, and subsequently at Aarhus, Denmark. The system was called SIS: Semantics Implementation System.

Robert Milne:
'Semantic relationships: reducing the separation between practice and theory'

Christopher Strachey believed that the gap between theory and practice was impeding the development of computing science. In my talk I shall consider how our work together on the essay that ultimately became our book tried to narrow the gap, by formalising, and reasoning about, the implementation concepts for programming languages. A particular focus will be the proof techniques for imperative programs that use storage, which were implicit, but not very easy to discern, in the book.

Philip Wadler:
'Christopher Strachey, first-class citizen'

The talk will review Christopher Strachey's influence on modern-day functional programming languages.

Uday Reddy:
'Parametric Polymorphism and Models of Storage'

In this presentation, we bring together two strands of Christopher Strachey's thought, *viz.*, *parametric polymorphism* and *abstract models of storage*. The term parametric polymorphism was introduced in [Str00], where Strachey distinguished it from "ad hoc" polymorphism. In the words of John Reynolds, "a parametric polymorphic function is one that behaves the same way for all types," whereas an ad hoc polymorphic function may have unrelated meanings at different types. [Rey83] A very similar intuition arose in mathematics, some twenty years earlier, for which Eilenberg and MacLane proposed a "General Theory of Natural Equivalences", what we now call category theory. [EM45] Relating the two notions allows us to develop a broad view of "parametricity" (the idea of acting the same way for all types), which

is applicable not only to programming languages but also to mathematics and perhaps other disciplines. A particularly important application of this broad view is for Strachey's "mathematical semantics" of programming languages.

For modelling imperative programming languages that operate by manipulating a store, Strachey presented a basic model of storage based on "locations", while promising a further paper on an "abstract model of storage" to appear in future. [Str97]. Unfortunately the latter never appeared. In succeeding work, Reynolds [Rey81] proposed an abstract model of store as well as an "intuitionistic" functor category approach to model the stack discipline of the store. O'Hearn and Tennent made the crucial observation that the model needs to be embellished with parametricity to capture the data abstraction aspects of the store. [OT95, TG00] In this talk, I review these developments as well as my recent work on integrating the Reynolds model with parametricity to capture the fact that state changes of the store are *irreversible*. [Red12, Red13]. One way to understand these developments is to view them as a "parametric mathematical semantics" of programming languages.

References

[EM45] S. Eilenberg and S. Mac Lane. General theory of natural equivalences. *Trans. Amer. Math. Society*, 58:231–294, 1945.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995. (Reprinted as Chapter 16 of [OT97])

[OT97] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages* (Two volumes). Birkhäuser, Boston, 1997

[Red12] U. S. Reddy and B. P. Dunphy. An automata-theoretic model of Idealized Algol. In *Automata, Languages and Programming (ICALP 2012)*, volume 7392 of LNCS, pages 337–350. Springer- Verlag, 2012.

[Red13] U. S. Reddy. Automata-theoretic semantics of Idealized Algol with passive expressions. In *D. Kozen and M. Mislove, editors, Proc. 29th Conf. on Math. Found. of Program. Semantics (MFPS XXIX)*, volume 298 of ENTCS, pages 325–348. Elsevier, 2013.

[Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North- Holland, 1981. (Reprinted as Chapter 3 of [OT97]).

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513– 523. North-Holland, Amsterdam, 1983.

[Str97] C. Strachey. The varieties of programming language. In *Algol- like Languages* [OT97], chapter 2. (original Tech. Report. Oxford PRG-10, 1973).

[Str00] C. Strachey. Fundamental concepts in programming languages. J. *Higher-order Symbolic Comput.*, 13(1-2):11–49, 2000. (original lecture notes, Copenhagen, 1967).

[TG00] R. D. Tennent and D. R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13:119–129, 2000.

Jeremy Gibbons: 'What are types for?'

Types in programming languages are commonly thought of as a way of preventing certain bad things from happening, such as multiplying a number by a string. But this is only half of the benefit of types: it is what types are *against*. Types in programming languages are also what enable some good things to happen, such as selecting the right implementation of a heterogeneous operation like comparison or printing based on type information; this is what are types *for*. This ability is surprisingly powerful, and gives rise to a variety of highly expressive generic programming techniques. I will illustrate with some

examples based on the rank-polymorphic array operations introduced in Iverson's APL: not only does the type information prevent array shape errors, it is what directs the lifting of operations across array dimensions.

Exhibition items

Some information here has been taken from the catalogue of the Strachey archive at the Bodleian Library, collected by the Contemporary Science Archives Centre. This is available online at `http://www.bodley.ox.ac.uk/dept/scwmss/wmss/online/modern/strachey-c/strachey-c.html`.

1. Account of Strachey's life and work. Date: 1926–1964.
   Tabulated account of his life and activity, drawn up by Strachey.

   The information, itemised by year and by month, is compressed in Strachey's very tiny handwriting on to 2 and a half sheets of writing-paper. The earliest entry is for 1926, and the entries have been added in blocks of two or three minutes at a time, to the end of 1964.

   Personal and professional events are recorded, and the account is remarkable for the amount of information encompassed.

2. Pictures of Strachey. Dates: 1923, 1925, 1973.
   A photo of Strachey aged 6 lying by the side of the pool at Mud, the Strachey holiday home in Sussex. Another of Strachey aged 8 wearing a hat and carrying a forkful of hay. Four photos of Strachey looking relaxed reading a newspaper in 1973, probably also taken at Mud. One computer generated image of Strachey, composed of various symbols including a moustache of λs.

3. Diagrams of the school gramophone at St. Edmund's School, Canterbury. Date: late 1940s.
   Strachey was employed as a schoolmaster at this small public school, where he taught physics and mathematics. He was also involved in extra-curricular school activities, including the school's radio club, where he designed and built a combined radio and gramophone.

4. 'Interplanetary travel'. Article in Harrow School magazine. Date: 1950.
   After leaving St. Edmund's school, Strachey took a post at the rather more prestigious Harrow. Here as well he took part in a number of activities, including teaching musical appreciation, and often gave lectures at the science society, including on the topic of interplanetary travel.

5. Items related to the 'Draughts' program. Date: 1950–1952.
   During the last years of his time at Harrow, Strachey became interested in computers, and spent some time at the National Physical Laboratory learning about their Pilot ACE machine. He then spent quite a few years working on a program which would enable a computer to play draughts.

6. Letter to Alan Turing. Date: May 1951.
   Strachey's copy of his letter to A.M. Turing, 15 May 1951. Written immediately after hearing Turing's talk 'Can Digital Computers think?', broadcast on BBC Third Programme on same day. In his letter Strachey discusses the general problems of 'thinking machines', their application to the Manchester machine, and his modification of his ACE draughts program.

7. 'Specification of the machine'. Date: 1957.
   Whilst working at the National Research Development Corporation, Strachey was responsible for large parts of the design of Ferranti's new computer, the Pegasus.

8. 'Time sharing in large fast computers'. Date: 1959.
   Typescript abstract, and 2 typescript versions of a paper read at the UNESCO Conference on Information Processing, Paris, July 1959.

   Strachey describes the paper as follows: "'Time Sharing in Large Fast Computers' was probably the first paper to discuss time-sharing and multi-programming as we now know them. It is a matter of history that the time-sharing idea became extremely fashionable in the middle sixties and dominated much of the work on computing at that time. When I wrote the paper in 1959 I, in common with everyone else, had no idea of the difficulties which would arise in writing the software to control either the time-sharing or multi-programming. If I had I should not have been so enthusiastic about them."

9. 'CPL working papers' (Introduction). Date: 1966.
   The CPL Working Papers were issued in July 1966 following a meeting of the CPL authors in Oxford on 24 June. They consisted of 2 parts: 1. The second edition of the Elementary

Programming Manual and 2. An unfinished Reference Manual. The concluding paragraphs of the minutes of the 24 June 1966 meeting state: "The circulation of the CPL Technical Report will conclude the phase of effort on CPL which has continued for the past four years, and at this stage most authors will probably wish to discontinue their efforts in a formal way... CPL of course needs major further work, particularly on data structures and program segmentation. C.S. (Strachey) and D.P. (Park) propose to study this at Oxford, and M.R. (Richards) may implement it at M.I.T."

Strachey's preface to the Working Papers, which gives a useful account of the history of CPL and its documentation, is on display here.

10. 'A general purpose macrogenerator'. Date 1965.
In an autobiographical note written in 1971, Strachey described this as follows:

"An interesting by-product of the work on CPL and its compiler was the General Purpose Macrogenerator which is in effect a simple but very general string processor. This program has now been implemented in almost every country in which computing is a research subject—I certainly know of implementations in Australia, Japan, Poland, Norway, Denmark, Germany, France, Italy, Holland as well as several in England and America. It gives an elegant example of the use of a stack and shows the power of functional composition to construct complicated operations from simple primitives. It is not, however, of any very deep theoretical interest and I now regard it as a rather beguiling time-waster."

11. Time sharing patent. Date 1965.
Strachey's work on time sharing culminated in US and UK patents, filed at the end of the 1950s and only awarded in the mid 1960s.

12. 'Systems analysis and programming', in *Scientific American*. Date: 1970.
Strachey described this as 'a popular article ... it describes, amongst other things, a draughts-playing program'. Strachey found difficulty in writing, and in adequately illustrating, the article, of which many drafts, revisions and alternatives survive in the folders both of working papers and editorial correspondence.

13. Single-page language semantics. Date: early 1970s.
One of the features of Strachey and Scott's mathematical semantics which must have particularly pleased Strachey is the compactness with which languages could be defined. This is an example of such a single page (two-sided) definition.

14. 'A theory of programming language semantics'. Draft manuscript (with Robert Milne). Date: 1974.
Strachey was determined to be elected to the Royal Society towards the end of his life, and partly in an attempt to increase his publication count and win some notoriety, he co-wrote with one of his students this extremely lengthy essay and submitted it for the Cambridge Adams prize. It represents the ultimate achievement of Strachey's work on semantics, and explains the concept and metalanguage as well as giving exhaustive worked examples on a sizeable programming language. The Strachey archive contains a large quantity of manuscript material, written in both Strachey's and Milne's hands. Displayed here is only the introduction and the note to the assessor's apologising for the length of the essay.

15. 'A theory of programming language semantics'. Draft typescript (with Robert Milne). Date: 1974.
A preliminary typed version of the essay, although still not completed. Strachey continued to revise the essay extensively, and there were plans to publish in 1975, although these were changed by Strachey's sudden death. The submission did not win the prize.

16. *A Theory of Programming Language Semantics*. Book, two volumes (Robert Milne and Christopher Strachey). Date: 1977.
After Strachey's death, Milne rewrote the material of the Adams Essay and it was published in book form.

17. 'The word games of the night bird'. Date: 1974.
An interview with Nancy Foy, published in *Computing Europe*.

18. *Eminent Victorians*. Book, Lytton Strachey (Christopher's uncle). Date: 1918.
Biographies of four leading figures from the Victorian era. Made Lytton Strachey's name as a biographer, and was notorious for the irreverence with which Strachey challenged the values and vices of the period. Lytton Strachey was one of the members of the hugely influential Bloomsbury group, which also counted amongst its members Virginia and Leonard Woolf, E. M. Forster, and John Maynard Keynes. Aside: Christopher's mother Ray's younger sister Karin married Adrian Stephen, Virginia Woolf's younger brother.

19. *Remarkable Relations: The Story of the Pearsall Smith Women*. Book, Barbara Strachey (Christopher's sister). Date: 1980.
Barbara who became a writer after her brother's death, writing on, among other things, the works of Tolkein. This book describes the lives of their mother's family line, and includes biography of Alys Pearsall Smith, first wife of Bertrand Russell.

Appendix: papers

Dana S. Scott. Greetings to the Participants at "Strachey 100". 2016.

Martin Richards. Christopher Strachey and the development of CPL. 2016.

Greetings to the Participants at "Strachey 100" by Dana S. Scott

I am very sorry indeed not to be able to make the trip from Berkeley, California, to be at this birthday party. And it is certainly hard to imagine that Christopher would be 100 years old this week! It is even harder to think that he died over 40 years ago at the age of 58 in May of 1975. I can still remember the deep, deep pain I felt in getting the not unexpected, but terrible news from the hospital, and the pain felt by all his circle of relatives, friends, students, and colleagues.

On my recent lecture tour in the UK last summer, I remarked that if one lives long enough, ***then*** one becomes something of "an historical figure"! And that is rather how I feel today speaking to you at age 84. People say, *"Oh, you knew Christopher Strachey!"* Or ask, *"You really met Kurt Gödel?"* Or, *"You really once visited Bertrand Russell?"* Or, *"What was Paul Erdős like?"* And though I never had a chance to meet Alan Turing (1912–1954), I met many, many people who did, including my colleague and good friend Robin Gandy (1919–1995). He was Turing's only Ph.D. student, remember, and, sadly, he also is now gone for over 20 years.

Let me give you all a ***warning*** and, perhaps, some ***advice***. Having met so many remarkable people in my career, I have that very unpleasant reaction—in reading their obituaries, for example—of asking myself: "Why in heaven's name didn't you ask them more questions about their lives?" We are often all too content with quite trivial, social conversations, but a pointed question will sometimes elicit fascinating stories. Now, a remarkable Oxford figure, Sir Isaiah Berlin (1909—1997), who was a favorite guest of ours when we lived in Oxford, never needed any prodding and was always full of good stories. Sometimes maybe Sir Isaiah indulged too much in what I like to call ***creative remembering***, but his entertainment value was always high. The lives of others, however, are often hidden, and they are the shy ones needing our encouragement to reminisce. So my advice is this: *Don't miss the chance of asking questions! It is much too easy to be too late.*

Fortunately Berlin has an extensive biography by Michael Ignatieff, and Strachey has a quite detailed, briefer biography by Martin Campbell-Kelly. A very detailed history of the Strachey family was written by Barbara Caine; while Christopher's sister, Barbara Strachey, has well documented the "remarkable" female side of the Stracheys and their relations. I mention some of these references in my bibliography.

What, we can wonder, would Christopher have been like at age 100? What indeed. Well, assuming he would have retained all his faculties, he definitely would have been ***outspoken***, ***critical***, and ***funny***—in other words he would have been quite the same as he was in his late 50s! But, perhaps, today as an ancient, respected guru, he would have allowed himself to be rather more ***forgiving***. We can only conjecture.

Of course, the whole aspect and style of the computing profession has completely changed over these last 40 years. We need to remember that Christopher was ***not*** in favor of an undergraduate degree. He was ***not*** in favor of the term "Computer Science". And he ***insisted*** on being called "Professor of Computing". And so it is somewhat difficult to think of how his attitudes would have changed now that we have something called ***machine learning*** winning Go games and making money in the stock market (to name only two recent areas of development that scare us).

And would he have liked ***the cloud*** and the extensive remote use of software? (I hope so.) Would he have approved of students getting an undergraduate degree and then opting for the large salaries rather than continuing to a Masters or even a D.Phil.? (I hope not, and I don't.) But too much is out of the hands of us academics today, and we have to learn to adjust. I think Christopher would have adjusted eventually, but he would have made us ***debate*** the changes — because he liked to force people to think.

Let us note that in this year, 2016, the CWI, Amsterdam, will also celebrate the 100th anniversary of Adriaan van Wijngaarden (1916–1987). He was the irrepressible force behind the design of ALGOL 68. One of van Wijngaarden's early students was Edsger W. Dijkstra (1930—2002), who received his Ph.D. in 1959. I knew Dijkstra and always found his highly critical and judgmental attitude hard to deal with. Aad van Wijngaarden was not at all like that, though he had great self-confidence. One of his later students, Jaco de Bakker (1939–2012), who received his Ph.D. in 1967, I liked enormously. Jaco had

a long and distinguished career as a researcher and teacher, both at the CWI (Centrum Wiskunde & Informatica) in Amsterdam and at the Vrije Universiteit.

I met van Wijngaarden and de Bakker in Amsterdam during the academic year 1968/69 while on sabbatical from Stanford University. Here is my favorite story about van Wijngaarden. The design of ALGOL 68 had been concluded and the report on the language came out the next year. When I received it back in the States in 1970 I was amazed by its complex typesetting, for it had been completely typed on an IBM Selectric typewriter using at least a dozen "golfballs" to get all those special symbols and typefaces. On my next visit to Amsterdam when I met van Wijngaarden I said to him: *"Aad, Aad! How in the world did you get a secretary to type such a complicated document?"* He smiled and answered: *"Oh, I have a secretary who types whatever I want!"* And he silently pointed to himself! Yes, he alone typed it all so he would know it was correct. That takes a truly remarkable determination.

Back in the early 1960s at Berkeley I had been introduced to ALGOL 60 and was very intrigued by its high-level design, having as a graduate student at Berkeley and Princeton done some (quite small-scale) machine-language programming projects. Then in the mid 1960s at Stanford, when the Computer Science Department was just being formed, I had many connections with the new group of students there and with John McCarthy (1927–2011). McCarthy had received his Ph.D. in 1951 at Princeton, not in Logic but in ***Partial Differential Equations*** under the direction of Solomon Lefschetz (1884–1972). He was however influenced by what he learned of the λ-calculus of Alonzo Church (1903–1995), another Princeton professor at the time.

McCarthy later introduced ***functional abstraction*** into his design of LISP, but on his own admission he did not understand much of Church's logical development of the λ-calculus. I, on the other hand, had as an undergraduate about 1952 at Berkeley learned about the subject on my own from a small, but rich logic text by Paul C. Rosenbloom (1920–2005), and I subsequently studied the monograph of Church (1941), the works of his close colleague Haskell Brooks Curry (1900–1982), and of his prominent students, Stephen Kleene (1909–1994) and J. Barkley Rosser (1907–1989), all of whom I got to know well in later life. However, during my time at Berkeley and Stanford I did not see the role λ-calculus would grow to play in programming-language theory.

During the spring of 1969 de Bakker and I worked on what we called ***"A Theory of Programs"*** defined in the spirit of Automata Theory, but with ***input*** and ***output***. I reported on the ideas when visiting the Vienna IBM Laboratory later that summer. The notes, handwritten by me, were finally reproduced in 1989 in a Festschrift for Jaco. That visit to Vienna was the start of many long-lasting friendships with the group there, who are now scattered around Europe.

But, more fatefully, I attended my first meeting of the IFIP Working Group WG 2.2 ***Formal Description of Programming Concepts***. I had been proposed for membership by my long-standing friend and mentor, Patrick Suppes (1922–2014), Professor of Philosophy at Stanford. He was unable to attend that meeting and sent me in his place. Alas, I have lost track of all my files, but some of the history of WG 2.2 is available on the internet. Here is the outline of the very early days:

- IFIP Working Conference on Formal Language Description Languages, met near Vienna in Baden-bei-Wien, 15–18 September 1964, organized by Prof. Heinz Zemanek (1920–2014), leader of the IBM Laboratory. This was the seminal event for the creation of WG 2.2.
- The first meeting of WG 2.2. was held at Alghero, Italy, 6–8 September 1967 organized by Prof. A. Caracciolo di Forino. The first chairman, until the Geneva meeting, was T. B. Steel Jr.
- The next meetings were at Copenhagen, Denmark, 22–26 July 1968, and then at Vienna, Austria, 21–25 April 1969, organized by Dr. Kurt Walk of the IBM Laboratory.

The April 1969 meeting was where I first met Christopher, and I was immediately impressed by the clarity of his statements. These working group conferences always had a lot of arguing back and forth (though not as bad as what I heard about the ALGOL meetings in WG 2.1!), and Christopher was a very good debater. His experience with the sadly failed CPL (Cambridge Programming Language) project, and his subsequent work, especially in collaboration with Peter Landin (1930-2009), had made him very aware of good design decisions. The ALGOL language had originally been only meant as a ***publication*** language

for numerical algorithms and was not originally intended to become a programming language. However, its clean ideas attracted many groups to make it—or something like it—an operational language. I note in passing that I was at a very well attended 40th Anniversary Meeting of WG 2.2 held in Udine, Italy, in September of 2006.

Let us now return to λ-calculus and Strachey's use of it. Christopher told me once that Roger Penrose (now Sir Roger!) suggested to him that he ought to look into using the λ-calculus for the kind of function definitions he wanted to use. At this moment I cannot track down or verify the story. (Perhaps people in Oxford might ask Penrose personally about this?) But, in Landin's three papers in 1964 and 1965, Peter from the start brings in λ-calculus—but without attribution to Strachey, even though they were close associates. It is of course quite possible that Landin thought of this himself, and he mentions Rosenbloom in his references. Landin in one abstract says:

> This paper is a contribution to the "theory" of the activity of using computers. It shows how some forms of expression used in current programming languages can be modelled in Church's λ-notation, and then describes a way of "interpreting" such expressions. This suggests a method of analyzing the things computer users write, that applies to many different problem orientations and to different phases of the activity of using a computer. Also a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations.

And at the end of his third paper he writes:

> This paper was written while I worked for Christopher Strachey. It is based on some lectures that George Coulouris invited me to contribute in Spring 1963 to a series on the "Logical Foundations of Programming" at the University of London Computer Unit. W. H. Burge, C. Strachey and M. Woodger read parts of an earlier version and pointed out many errors and obscurities.

So, there is no question that many ideas were shared. In any case, when I met Strachey and subsequently arranged to spend the Autumn term in Oxford in 1969 working with him, he was using λ-calculus in the style of Landin's papers, which were much read by all of us wanting to follow the paths Landin had convincingly set forth.

Form the beginning of our association in Oxford I told Christopher, ***"Lambda-calculus has no mathematical models!"*** Calling on my background in model theory and recursion theory, I suggested a partial-order structure for functions and functionals that had been much employed could be adapted to his objectives. There were several earlier works doing computability theory in this way, though at that time the theory people were much more involved in understanding ***infinitary*** computation. Using inspiration from the Ph.D. thesis of Richard Platek and writings of Kleene, I suggested a formulation for Strachey to use in my draft paper *"A type-theoretical alternative to ISWIM, CUCH, OWHY."*

Then, one Saturday morning in November of 1969, when lying on the bed in the guest room of the flat that I and my family had rented in Headington, an awful thought occurred to me. In setting up the higher types of continuous functionals, I had seen that each of the type domains had a ***basis of finite*** elements—in the sense that each object of each type is the least upper bound of the finite elements it contains. In passing from one type level to the level of continuous functions above it, the basis of finite elements usually became ***more complicated***. But what if, I suddenly thought, there were a type D such that the basis of the function space $(D \to D)$ was ***no more*** complicated than the basis of D? Could we then have a type $D = (D \to D)$ in the sense of an isomorphism of partially ordered sets? In a fever I set about finding such a domain over the next few days, and that was how Domain Theory was born in my mind. After being so critical of the formal approach to λ-calculus, I would now have to eat my own words, because the λ-calculus could indeed have ***mathematical*** models.

In many ways I rather regret having cast the foundations of Domain Theory at that time in the form of ***lattice theory*** (and later ***DCPO theory***). Some people liked it, and some people disparaged it

as ***"Scottery"*** and too abstract. A change came in the mid 1970s when I had become a colleague of Christopher's at Oxford as the first Professor of Mathematical Logic. Gordon Plotkin, then a new Ph.D. in AI at Edinburgh, sent me a 1972 memo, *"Set-theoretical and other elementary models of the λ-calculus,"* inspired by Domain Theory. I at first did not fully understand Plotkin's simplification (by focussing on one particular domain), and I put the memorandum aside. But later for teaching purposes I went back to it and realized that, by using ideas of Recursive Function Theory, the set theory could be replaced just by the ***powerset*** of the set of natural numbers. I wrote up a report on the approach in 1975, which was published the next year, called, "Data types as lattices: To the Memory of Christopher Strachey, 1916–1975." That paper had considerable historical detail and suggestions about understanding higher types. Of course, by then Christopher was gone. My earlier unpublished memo on the type-theoretical alternative, and Plotkin's memo on the set-theoretical model were finally published in the Festschrift for our friend and colleague Corrado Böhm in 1993—along with additional materials.

And here is another deep regret of mine. I was a graduate student at Princeton in Mathematics from 1955 to 1958 working under the direction of Church. Now he never spoke to us students at that time about λ-calculus. My conjecture is that he was always deeply disappointed that his early idea about a new foundation for logic and mathematics proved to be inconsistent, as his students Kleene and Rosser discovered. He wrote up the ***equational part*** of the theory (proved consistent by syntactical arguments) in 1941 and then turned to ***typed*** λ-calculus, and this later work became very influential indeed. Curry, however, continued to do research in the ***untyped*** theory all his life, while Rosser and Kleene turned away from it after their graduate school period.

What then is my regret? Well, in 1955 John R. Myhill (1923–1987) and John Shepherdson (1926–2015) published their paper *"Effective operations on partial recursive functions"*. And in 1956 Richard M. Friedberg and Hartley Rogers Jr. (1926–2015) wrote an abstract on what became their 1959 paper, *"Reducibility and completeness for sets of integers"*. Friedberg and Rogers called their operators ***enumeration operators***, but they were basically the same as Myhill and Shepherdson's functionals. And it took me almost 20 years ***until 1975*** to see that they had therefore ***already discovered*** a model for the λ-calculus—but none of them ever seemed to know it!

Since I knew Myhill and Rogers personally and had read Shepherdson's other work, and since I had a good background in recursive function theory and model theory from my student days, and since Kleene was resident in Princeton for 1956/57, ***and*** since Rodgers visited Princeton that academic year to lecture, ***I myself could have put two and two together to produce a model in 1957!*** If only I had done so and then announced it at the big 1957 Summer Logic Conference at Cornell University, I would now be ***rich*** and ***famous*** (though the famous part is not my principal regret). Also the development of the theory of programming languages would have been much different. Alas, history is not an experimental science, and we cannot turn the clock back to correct our oversights and erase our stupidities.

Another regret is that my taking up the Oxford chair did not start up further collaboration with Strachey. There were two reasons. For better or worse Oxford is governed by committees of academics. Strachey had been given a personal chair in the early 1970s, and my chair was in both Philosophy (under the old Lit Hum Board) and in Mathematics (with its own board). That meant both Christopher and I had to take part in many committee meetings. Also the eight-week Oxford terms went by at a lightning pace. There were also many students around (fortunately!) requiring supervision, but one term owing to leaves of absence I had to supervise 19 postgraduates.

But a second reason that limited collaboration was the decision Strachey made to submit a book to Cambridge for the Adams Prize. The rule was that all authors had to be former Cambridge students or fellows, and so, as I could not be a co-author, Christopher recruited Robert Milne. The resulting Strachey-Milne opus was perhaps too dense and too technical, and in the event it failed to garner the prize—a sad, disheartening fact Christopher learned in hospital just before he died.

Fortunately, Strachey's loyal assistant, Joseph E. Stoy, stepped up to the task and wrote "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory." That was an excellent presentation, but it would have perhaps been even better with three authors.

At the time of my discovery of the λ-calculus model in 1969, John Reynolds (1935–2013) was visiting

England and attended my first lectures about the approach. He became an enthusiastic advocate, as he had been working himself on programming-language design. Over the years he subsequently made many major contributions to design and theory, as recounted in the Brookes–O'Hearn–Reddy Reynolds memorial paper in 2014.

Reynolds, in works with his Ph.D students and, later, with Peter W. O'Hearn, brought a fresh new approach to Denotational Semantics. In the memorial paper the authors write:

> These type constructors are parametrized by a type variable S representing the set of states for a store. This is in marked contrast to Strachey's denotational semantics where the store is fixed and global. Here the store is a type variable. In essence, this means that the program can work with whatever "type" of store we supply, as long as it has all the required data variables. An intuitive interpretation of parametric polymorphism immediately implies that the program will act the same way for all these different "types" of stores.

Ah, if only that could have been discovered in the 1970s, the influence of Denotational Semantics would have been much greater, as the original Strachey approach to modeling the store was quite *ad hoc* and often unconvincing. Fortunately, however, research continues vigorously today. Most recently O'Hearn has been co-recipient with Stephen Brookes of the 2016 Gödel Prize for the invention of Concurrent Separation Logic, which was deeply influenced by Reynold's ideas and results.

When I was invited to speak at the Princeton University and the Association for Computing Machinery's Alan Turing Centennial Celebrations in 2012 I tried to make a ***time line*** of events and ideas in Logic, Computability, and Category Theory. This is only a faint approximation to a history of ideas, but that might encourage others to flesh out my outline. I also just found through an internet search the slides of a talk by Pierre-Louis Curien on the semantics of programming languages. He and the French have made major contributions to the area, and his report might also be an inspiration to historians.

In conclusion, then, we can agree this has been an amazing half century in the development of computers and the ways of using them. New surprises pop up every week. Ten years ago we could not have predicted even half of what is available today. And, of course, I am deeply sad that Christopher could not live to see what has happened.

Thanks, and my best greetings this November of 2016.

A VERY SELECT BIBLIOGRAPHY **1941**
Alonzo Church, "The Calculi of Lambda Conversion." Princeton University Press, 1941, 77 pp.

1950
Paul C. Rosenbloom, "The Elements of Mathematical Logic." Dover, 1950, iv + 214 pp.

1964
Peter J. Landin, "The mechanical evaluation of expressions." The Computer Journal, vol. 6 (1964), pp. 308–320.

1965
Peter J. Landin, "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". Communications of the ACM, vol. 8 (1965), pp. 89–101.

Peter J. Landin, "A correspondence between ALGOL 60 and Church's Lambda-notation: part II". Communications of the ACM, vol. 8 (1965), pp. 158–165.

1966
Peter J. Landin, "The next 700 programming languages". Communications of the ACM, vol.9 (1966), pp. 157–166.

1967
Christopher Strachey, "Fundamental concepts in programming languages". Unpublished lectures in Copenhagen (August 1967) – finally published in Higher-Order and Symbolic Computation, vol. 13 (2000), pp. 11–49.

1969
J.W. de Bakker and Dana Scott, "A theory of programs: An outline of joint work." Handwritten notes presented at an IBM Seminar, Vienna, August 1969. Reprinted in: J.W. Klop, J.J.C. Meijer, and J.J.M.M. Rutten (eds.), "J.W. de Bakker, 25 Jaar Semantiek: Liber Amicorum." CWI Amsterdam, 437 pp.

Dana S. Scott, "A type-theoretical alternative to ISWIM, CUCH, OWHY." Informally distributed, October 1969. Published with additions by the author in: Theoretical Computer Science, vol. 121 (1993), pp. 411–420.

1972
Gordon Plotkin, "Set-theoretical and other elementary models of the λ-calculus." Theoretical Computer Science, vol. 121 (1993), pp.351-409, including the previously unpublished 1972 memorandum.

1976
Dana Scott, "Data types as lattices: To the Memory of Christopher Strachey, 1916-1975." SIAM Journal of Computing, vol. 5 (1976), pp. 522-587.

1977
Joseph E. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory." MIT Press 1977, xxx + 414 pp.

1980
Barbara Strachey, "Remarkable Relations: The Story of the Pearsall Smith Family." London, Victor Gollancz, 1980, 351 pp.

1985
Martin Campbell-Kelly, "Christopher Strachey, 1916-1975: A Biographical Note." IEEE Annals of the History of Computing, vol. 1 (1985), pp. 19-42.

1990
C.A. Gunter and D.S. Scott, "Semantic Domains." In: "Handbook of Theoretical Computer Science, Volume B," edited by J. van Leeuwen, North Holland, 1990, pp. 633-674.

1998
Michael Ignatieff, "Isaiah Berlin: a life." Metropolitan Books, 1998, 356 pp.

2000
Dana Scott, "Some Reflections on Strachey and His Work." Higher-Order and Symbolic Computation, vol. 13 (2000), pp. 103–114.

Roger Penrose, "Reminiscences of Christopher Strachey." Higher-Order and Symbolic Computation, vol. 13 (2000), pp. 83–84.

2005
Barbara Caine, "Bombay to Bloomsbury: A biography of the Strachey family." Oxford University Press, 2005, xvii + 488 pp.

2009
Richard Bornat, "Peter Landin: a computer scientist who inspired a generation, 5th June 1930 - 3rd June 2009." Formal Aspects of Computing, vol. 21 (2009), pp. 393–395.

2011
Pierre-Louis Curien, "A journey into the semantics of programming languages." Slides for a talk on 24 May 2011 at the Institute of Software of the CAS, Beijing, China, 49 pp.

2012
Dana S. Scott, "A Timeline for Logic, λ-Calculus, and Programming Language Theory." Slides for talks prepared for: TURING CENTENNIAL CELEBRATION, Princeton University, May 10-12, 2012, and ACM TURING CENTENARY CELEBRATION, San Francisco, June 15-16, 2012, 9 pp.

2014
Dana S. Scott, "Stochastic λ-calculi: An extended abstract." Journal of Applied Logic, vol. 12 (2014), pp. 369–376.

Dana S. Scott, "Types and Type-Free λ-Calculus." Slides for various lectures, 28 pp.

Stephen Brookes, Peter W. O'Hearn, and Uday Reddy. "The essence of Reynolds." Formal Aspects of Computing, vol. 26 (2014), pp. 435-439.

Postscript: This talk was not meant as a complete history. There are easily 150 other names that ought to be added to a description of the development over half a century of programming-language definition and semantics — and offshoots from that research. The present author is incapable of writing such a history and only wanted to trace some of the paths he was personally involved in that were directly inspired by Christopher Strachey.

Christopher Strachey and the Development of CPL

by

Martin Richards
mr@cl.cam.ac.uk
www.cl.cam.ac.uk/users/mr10

Computer Laboratory
University of Cambridge
Mon Oct 17 08:16:40 BST 2016

Abstract

Chrisopher Strachey was the most significant contributor to the design and implementation of the programming language CPL. At the time there was little understanding of the complexities of computer language design and how type systems could cope with lists and the kinds of structures needed to represent, for instance, parse trees. The CPL project cannot be regarded as being successful since it did not result in a usable CPL compiler. The reasons being that the language became too large and complicated, there were insufficient people to implement the compiler and, in the middle of the three year project, all work had to be transferred from Edsac 2 to Titan, a newly designed version of the Ferranti Atlas computer which as yet had no operating system. Even so, we can be proud of the work that went into CPL and its influence on the design of many later languages.

Keywords: Edsac, Titan, CPL, BCPL

1 Introduction

Before discussing the development of CPL it it necessary to have some understanding of what computer facilities were available at Cambridge at the time.

The first electronic computer in Cambridge was constructed under the Direction of Maurice Wilkes and successfully ran its first program in May 1949. It was a serial machine with up to 512 35-bit words of memory stored in mercury delay lines. Each word could contain two 17-bit instructions. Wilkes made two important decisions that contributed to the machine's success. Firstly, he chose to use a fairly slow clock rate of 500kHz arguing that reliability and ease of design were more important than speed since the machine was already going to be able to do calculations thousands of times faster than was possible by hand. Secondly, he made the machine available to anyone in the University who could make use of it. This was enthusiastically taken up by many departments including the radio astronomers, the theoretical chemists, and mathematicians, directly contributing to at least two Nobel Prizes.

One advantage of the serial design was that it allowed high precision arithmetic to be implemented using rather few logic gates. Although the instruction set was quite primitive with no

subroutine jump instructions and no index registers, it did include an instruction to add the product of two 35-bit numbers to a 71-bit accumulator. Indeed, it was a good machine for serious numerical calculation. For more information look at the poster[12] which is on the wall of the Cambridge Computer Laboratory's first floor Coffee Room describing how the incredibly cunning EDSAC initial orders worked, written by David Wheeler. In a mere 31 instructions, it could load a program from paper tape written in assembly language and execute it.

In 1958 the EDSAC was replaced by Edsac 2. This was a bit sliced computer controlled by a micro program consisting of a matrix 32x32 ferrite cores. Each core corresponded to a different micro instruction. When a core was fired, windings on it would give the (x, y) address of the next micro instruction to obey and other windings would send signals to control the machine. Some micro instructions used signals from the machine, such as the sign of the accumulator, to modify the address of the next core to fire. The micro program consisted of up to 1024 instructions many of which were conditional. David Wheeler wrote the micro program and as with the Initial Orders of the EDSAC, it allowed paper tapes of assembly language to be loaded and executed. But unlike the EDSAC, the instruction set was much more powerful and convenient to use and included some rather wonderful debugging aids. For instance, by pressing a switch on the control panel, the machine would output a paper tape giving a compact represention of the recent flow of execution of the program, typically in the form of recently executed nested loops. Another intriguing feature was that the fifth bit of the effective address of every instruction was fed to a loudspeaker normally producing a rich sound that programmers soon learnt to interpret. For users not using assembly language the preferred language was Edsac Autocode, designed and implemented by David Hartley.

Initially, Edsac 2 had a memory of 2048 20-bit words. These were used to represent machine instructions, but some instructions used pairs of these words to represent 40-bit integers or floating point numbers. The machine also had a read only memory filled with many useful subroutines, typically entered by the instruction with function code 59. The main memory was extended in 1962 by an additional 16K words purchased from Ampex. Access to this memory was less convenient since it required the use of a new base register.

Around 1961, it was clear that the University needed a more powerful computer, and it was decided to install a version of the Ferranti Atlas, but, to save money, a cut down version was designed. Following the success of the previous two computers, it is perhaps not surprising that modifications to the machine were designed and built in Cambridge. Some users would probably have preferred a more conventional machine such as the IBM 7094 allowing greater compatibility with other universities, particularly in America.

David Wheeler had a major hand in the modified hardware design. Since the machine was going to allow simultaneous access for many users, it required hardware memory protection, but rather than the full blown paging mechanism of the full Atlas, David designed a much simpler system using a base address and limit register. For efficiency reasons, the base address was ORed into the effective address rather than being added. This meant that with a main memory of 32K words, small programs of say 4K could be placed in 8 positions in memory but programs larger that 16K only had one choice. To make time sharing responsive, small programs were strongly encouraged. This cut down machine was called Titan but was not ready to use until 1964 and even then it needed an operating system, an assembler and at least one high level programming language. With the limited resources available, an assembler was constructed on time but the operating system was clearly going to be late, and the decision to design and implement CPL which was substantially more complicated than ALGOL 60 was extraordinarily optimistic. At a late stage, Peter Swinnerton-Dyer was called in and miraculously implemented, in about six weeks,

a simple operating system including an assembler, and a version of Edsac Autocode. Edsac 2 was switched off on 1 November 1965.

2 Development of CPL

The initial plan was to design a new language related to ALGOL 60, taking advantage of its many good features such as its block structure, but removing or modifying features that caused its implementation to be difficult or inefficient. A paper by Strachey and Wilkes[21] proposed some possible improvements. One idea was to allow programmers to state when functions were non recursive and another was a scheme that attempted to stop functions having side effects to allow compilers more freedom to optimise the order of evaluation within expressions. They also felt that a simpler form of ALGOL 60's call-by-name would be useful. They called this call-by-simple-name being equivalent to call-by-reference we use today. These ideas found their way into the design of CPL. The work on CPL ran from about 1962 to 1966. The first outline of CPL was in the paper "The main features of CPL"[2] published in 1963 and a later document "CPL Papers"[3] was written in 1966.

I was involved with CPL when I was a research student for three years starting in October 1963. By then CPL changed from being Cambridge Programming Language to Combined Programming Language since the project was now a collaboration with the University of London, Institute of Computer Science. The committee involved in the design of CPL was headed by Strachey and included his assistant Peter Landin, David Hartley, David Park, David Barron, from Cambridge and, from London, John Buxton, Eric Nixon and George Coulouris who implemented a satisfactory subset of CPL on the London Atlas computer. As a junior research student involved with the Cambridge implementation, I was lucky enough to attend most of these meetings. They usually took place either in Cambridge or in Strachey's house in Bedford Gardens, London. The discussions were lively and animated trying to reach compromises between the conflicting desires of the participants. Some wanted a firm underlying mathematical stucture to the design, others only wanted features that were useful, and those involved with the implementation favoured a design that could be compiled efficiently. In general, the kind of discussions were like those fictitous language design discussions appearing in chapter 8 of "System Programming with Modula-3" edited by Greg Nelson[10]. During breaks in the discussions, Christopher and Peter would sometimes entertain us with piano duets.

Clearly the designers of ALGOL 60 were familiar with Church's λ-calculus since the ALGOL 60 Report describes procedure calling and call-by-name parameter passing in term of textual replacement of procedure calls by suitably modified copies of the procedure bodies, in more or less the same way that λ-calculus used α-conversion and β-reduction. This was clarified in Peter Landin's paper "Correspondence between ALGOL 60 and Church's Lambda-notation"[7]. With such a close relationship between ALGOL 60 and λ-calculus, it was natural to allow procedures to be defined within other procedures.

CPL was a natural extension of ALGOL 60 with a wider variety of numerical values, including integers, reals and complex numbers both in single and double precision as well as bit pattern values called logicals. It also contained a wide variety of useful and easily implemented conditional constructs using keywords such as `if`, `unless`, `while`, `until` and `repeat`. Strachey was always full of great ideas for other extensions, not all of which ended up in CPL. He was keen on the idea of L and R values and L and R mode evaluation. These corresponded to the different ways in which expressions on the left and right hand side of assignment statements were processed. He also

liked the idea of referential transparency which essentially means an expression can be replaced by any other expression that evaluates to the same result. This led to the belief that conditional expressions and even function calls should be allowed on the left hand side of assigments and in call-by-reference parameters. Strachey thought it important to make a distinction between functions that produced results and routines whose sole purpose was to have side effects. He liked the idea that some functions could be defined in such a way that they could not have side effects. He called them fixed functions and had a way of implementing them. Rather than use the common choice of dynamic and static chains through the runtime stack, he proposed the construction of a list of free variable values at the time a function was defined. For fixed functions, the values were R values and assignments to them would not cause side effects. Free functions on the other hand would use L values, allowing them to have side effects. His liking for L values extended to the idea of creating a general form consisting of a load-update pair (LUP), allowing a user to, for instance, construct the L value of a field of bits within a logical variable.

In addition to call-by-value and call-by-reference, CPL had ALGOL 60 style call-by-name using the keyword `subst`, but this was soon dropped since passing a function as an argument was a simpler way to provide the same facility.

One notable feature of CPL was its ability to deduce the data types of all expressions and variables with little explicit help from the programmer. Partly because we had insufficient understanding of type systems, we needed a type `general` which carried the actual type of values around at runtime. The much more recent language ML[9] has a superb polymorphic type system with the ability to statically deduce the types of all expressions and variables. The design of ML was a tour de force and was cited as one of the reasons why Robin Milner received the Turing Award in 1991. Although ML can deduce the type of every expression in an ML program, this is only just true because the process can take exponential time. For instance, the type of `f` in the following short program is many millions of characters long.

```
fun a x y = y x x;
fun b x   = a(a x);
fun c x   = b(b x);
fun d x   = c(c x);
fun e x   = d(d x);
fun f x   = e(e x);
```

3 GPM and the CPL Compiler

In order to write the CPL compiler in a form that could be easily transferred from Edsac2 to Titan, it was decided to implement the compiler in essentially a subset of CPL and hand translate it into a sequence of macro calls using the GPM macrogenerator that Strachey specifically designed for the purpose. GPM was described in detail in a paper in the Computer Journal[18]. Since GPM had such a significant influence on how the compiler was constructed it is necessary to learn a little about how GPM worked.

Unfortunately Strachey's paper contains a subtle bug as a result of his attempt to implement GPM using a single stack. I therefore present a re-implementation of GPM called BGPM that uses two stacks. BGPM is implemented in BCPL (not in CPL) using only ASCII characters rather than those available on the flexowriters used at Cambridge. Its behaviour is almost identical to GPM. In BGPM a typical macro call is: `[aaa,bbb,ccc]`. The character `'['` marks the start of a macro call,

`','` separates the macro arguments and `']'` indicates that a macro call is complete and should be expanded. BGPM's left hand stack holds incomplete macro calls while their arguments are being formed. On encountering `']'` the latest macro call is now complete and is moved to the right hand stack. This stack contains complete macro calls and also the environment chain of defined macros. The arguments of a macro are numbered from zero upwards, and when a complete call is moved to the right hand stack its zeroth argument is looked up in the environment causing input to be temporarily changed to the start of the body of the matching macro definition. An error occurs if the macro is undefined.

While executing the body of a macro, a substitution item of the form `#`n is automatically replaced by a copy of the n^{th} argument. Sometimes, especially when defining macros, it was necessary to disable normal processing of special characters, and this was done by enclosing the text in curly brackets (`{` and `}`) which can be nested. Finally, there was a comment character (`'`) which caused text to be skipped up to the first non-white-space character on a later line of input.

A new macro definition can be inserted at the front of the environment using the predefined macro `def` as in the following demonstration.

```
[def,hi,{Hello #1}]'
[hi,Sally]
[hi,James]
```

This would would add the macro `hi` with body `Hello #1` to the chain of defined macros and call it twice, generating the following result:

```
Hello Sally
Hello James
```

Another built in macro, `set`, allows the body of a macro to be updated, essentially making it behave like a variable, and with the aid of the `eval` macro, it is possible to perform integer arithmetic, as in:

```
[def,counter,0000]'
[def,inc,{[set,#1,[eval,[#1]+1]]}]'
[inc,counter]counter = [counter]
[inc,counter]counter = [counter]
```

This will generate the following.

```
counter = 1
counter = 2
```

A notable feature of BGPM is that, if macros are defined within an argument list of a macro call, they are removed when the macro finishes execution. Using this feature, it is easy to define conditional macros such as `ifeq`. BGPM is thus simple but remarkably powerful, allowing one to define, for instance, the macro `prime` to generate the n^{th} prime, so that `[prime,100]` would expand to `541`.

Some of the macros used in the Cambridge CPL compiler are exemplified by considering the implementation of the following CPL definition of the factorial function.

let rec `Fact(n) = n = 0 -> 1, n * Fact(n-1)`

The corresponding macro translation could be as follows.

`[Prog,Fact]`	The start of a function named `Fact`
`[Link]`	Store recursive function linkage information onto the stack
`[LRX,1]`	Load the first argument of the current function onto the stack
`[LoadC,0]`	Load the constant zero onto the stack
`[Eq]`	Replace the top two values on the stack by `TRUE` if they were equal and `FALSE` otherwise
`[JumpF,2]`	Inspect and remove the top value of the stack and jump to label 2 if it was FALSE
`[LoadC,1]`	Load the constant `1` onto the stack
`[End,1,1]`	Return from the current function returning one result in place of its single argument
`[Label,2]`	Set label 2 to this position in the code
`[LRX,1]`	Load the first argument of the current function onto the stack
`[LoadC,1]`	Load the constant `1` onto the stack
`[Sub]`	Replace the top two items on the stack by n-1
`[Fn,Fact]`	Call the function named `Fact`
`[LRX,1]`	Load the first argument of the current function onto the stack
`[Mult]`	Replace the top two values on the stack by the product of their values
`[End,1,1]`	Return from the current function returning one result in place of its single argument

To call a function, its arguments are loaded onto the stack in reverse order followed by the subroutine jump, typically `[Fn,Fact]`. On entry to the function, linkage information is pushed onto the stack by the `Link` macro. After evaluating the body of the function, a return is made using the `End` macro that specified how many arguments to remove and how many results to copy in their place from the top of the stack. It was helpful to use readable function names such as `Fact` in the macro code. Since the assembly language for both Edsac 2 and Titan only had numerical program labels the macros `Prog` and `Fn` would convert function names to label numbers using macros that mapped function names to label numbers.

The CPL compiler written in a subset of CPL was large, and it gave us a reasonable feel of what programming in CPL was like. Our subset of CPL needed a mechanism to allow us to construct hash tables, lists and structures such as the parse tree generated during syntax analysis. This was easily accomplished using a simple memory model in which all memory locations were of the same size and labelled (or addressed) by consecutive integers. Macros were added to return the integer addresses of arguments (`LLX`), locals (`LLW`) and globals (`LLG`), and a macro `RV` was added to access the contents of a memory location given its address.

We found we did not need to define functions within other functions. This allowed us to represent functions by just their entry points without any additional environment information. This also meant that function calls did not need to implement either Dijkstra displays or Strachey's free variable lists. It also allowed the compiler to be broken into several sections each compiled separately. We only needed called-by-value arguments, since pointers could be used for call-by-reference arguments, and call-by-name could be implemented by passing functions. It is worth noting that the CPL program given in Strachey's GPM paper only used call-by-value and never defined a function within another.

We found that we could survive with all values being the same size, since they could represent all the kinds of value we needed, including integers, characters, truth values, arrays and functions, and this greatly reduced the number of macros we had to define. It was just as well that functions were represented by a single pointer.

The resulting subset of CPL that was used is outlined in my PhD thesis[11] and it closely resembles BCPL, but we still felt we were programming in CPL.

Within the compiler, it was frequently necessary to write code to generate error messages and output assembly language. It was clear that the way GPM expanded macros by copying their bodies while replacing substitution items by arguments was ideal. This resulted in the definition of `writef` whose first argument was like the body of a macro but with slightly extended substitution items such as `%c`, `%s` and `%i5` to cope with characters, strings and numbers. There were as many additional arguments as needed and substitution items did not need to specify argument numbers since arguments were used in sequence. I regard `writef` as a direct result of our experience with Strachey's invention of GPM.

An extension of `writef` called `printf` is widely used in C, but due to its polymorphism it is difficult to implement it in strictly typed languages and so, sadly, is not available in ML or Java.

4 MIT, BCPL and PAL

Strachey visited Jack Wozencraft at MIT when he was designing a new course to introduce first year students to computer programming. The outcome was that both Peter Landin and I moved to MIT and helped to design a form of sugared λ-calculus, based on the language ISWIM (If you See What I Mean) described by Landin in "The next 700 programming languages"[8]. Landin produced a prototype implementation in LISP using his SECD machine as described in "The mechanical evaluation of expressions"[6]. When I arrived at MIT in December 1966, I quickly constructed a BCPL compiler that ran on the timesharing system CTSS and set about using it to implement a variant of Landin's system called PAL[1] that was then used in the new course. This language was a form of sugared λ-calculus but its syntax was somewhat related to CPL. Based on the SECD machine it was dynamically typed with every value carrying its data type at run time. Like LISP, it required a garbage collector. Many years later this course was superceded by one based on a cleaned up version of LISP called Scheme. The lecture notes for this replacement were turned into the book "Structure and Interpretation of Computer Programs"[4]. A similar course was later started in Cambridge, UK, using ML. BCPL still exists and is freely available[13] and has a manual[14].

5 OS6 and Tripos

Throughout my time at MIT and later, on return to Cambridge, I maintained close ties with Strachey. He was enthusiastic about the design and portability of BCPL to the extent that he implemented it on the KDF-9 computer at Oxford, and later he installed it on the CTL Modula One at the Programming Research Group at 45 Banbury Road. He also used BCPL to implement an operating system called OS6 for that machine which, as the story goes, ran successfully, for the first time, on a Sunday evening very few days after the machine arrived. The BCPL source code is published in OS6Pub[19] with the "Commentary on the text of OS6Pub"[20]. BCPL running

under OS6 was the only language available to students at the Programming Research Group for some years.

OS6 included many innovative ideas, one being his implementation of I/O streams. He represented a stream by a small vector, called a stream control block (SCB), that held all the values needed for its implementation, such as a pointer to its buffer and integers giving its size and current position. But importantly, the SCB also held three functions. One (`Next`) to read the next value from the stream, one (`Out`) to write a value to the stream and one (`Close`) to close the stream. `Next` was defined as follows:

```
LET Next(s) = (s!NEXT)(s)
```

where `s` points to an SCB and `NEXT` is a manifest constant identifying the position of the read function. This mechanism could be used to represent streams of characters, words or indeed any kind of value. You could even create a stream that generated prime numbers. It is clear that this structure is similar to an instance of a class in a language such as Java. Some streams required larger SCBs but could still use the same definition of `Next`. This is clearly a precursor to the concept of inheritance found in modern object oriented languages.

Some years later I led a team of research students that implemented a portable operating system called Tripos[16]. This was used for some years in Cambridge for research concerned with computer networks and the Cambridge Ring. BCPL running under a version of Tripos is still used commercially controlling the factory floor of many Ford car plants around the world. An interpretive version of Tripos called Cintpos[15] is still freely available.

6 Conclusions

I am deeply indebted to Christopher for the help and discussions we had when I was a Research Student and for arranging that I could go to MIT after my PhD. Without him and the CPL project, BCPL would not have been developed and it would not have been seen by Ken Thompson. Ken's language B[5] would not have been designed and his collaboration with Dennis Ritchie would not have resulted in the development of C[17] and it successors such as C++ and Java. Whether this was good is perhaps still debateable.

References

[1] A. Evans, Jr. PAL - A language designed for teaching programming linguistics. In *Proceedings of 1968 23 ACM Conference, pp 395-403 August 1968.* Thompson Book Company, Washington, DC, 1968.

[2] D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, and C. Strachey. The main features of CPL. *The Computer Journal*, 6:134–143, July 1963.

[3] Editor: C. Strachey. CPL Working Papers, V53-57. Technical report, Computer Laboratory, Cambridge University, July 1966.

[4] G.J. Sussman H. Abelson and J. Sussman. *Structure and Interpretation of Computer programs.* MIT Press, Cambridge, MA, 1985.

[5] S.C. Johnson and B.W. Kernighan. *The Programming Language B.* http://cm.bell-labs.com/who/dmr/bintro.html, 1997.

[6] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[7] P.J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation, Parts 1 and 2. *Comm. ACM*, 8(2 and 3), February and March 1965.

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

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

[10] Ed. G. Nelson. *System Programming with Modula-3*. Prentice Hall, 1991.

[11] M. Richards. *The Implementation of CPL-like programming languages*. PhD thesis, Cambridge University, 1966.

[12] M. Richards. *EDSAC Initial Orders and Squares Program* . www.cl.cam.ac.uk/users/mr/edsacposter.html, 2009.

[13] M. Richards. *The BCPL Cintcode Distribution*. www.cl.cam.ac.uk/users/mr/BCPL/bcpl.{tgz,zip}, 2011.

[14] M. Richards. *The BCPL Programming Manual*. www.cl.cam.ac.uk/users/mr/bcplman.pdf, 2011.

[15] M. Richards. *The Cintpos Distribution*. www.cl.cam.ac.uk/users/mr/Cintpos/cintpos.{tgz,zip}, 2011.

[16] M. Richards, A.R. Aylward, P. Bond, R.D. Evans, and B.J. Knight. The Tripos Portable Operating System for Minicomputers. *Software-Practice and Experience*, 9:513–527, June 1979.

[17] D.M. Ritchie. *The Development of C*. http://cm.bell-labs.com/who/dmr/chist.html, 1997.

[18] C. Strachey. A General Purpose Macrogenerator. *The Computer Journal*, 8(3):225–241, October 1965.

[19] C. Strachey and J. Stoy. *The text of OS6Pub*. Programming Research Group, Oxford University Computer Laboratory, July 1972.

[20] C. Strachey and J. Stoy. *The text of OS6Pub (Commentary)*. Programming Research Group, Oxford University Computer Laboratory, July 1972.

[21] C. Strachey and M.V. Wilkes. Some Proposals for Improving the Efficiency of ALGOL 60. *Comm.A.C.M.*, 4:448, 1961.