Generator and Search Objects in Java

Lloyd Allison
School of Computer Science
Monash University
Australia 3168
http://www.csse.monash.edu.au/

Generator classes are defined in the object oriented programming language Java by using continuation-style programming. Generator objects are used to write Prolog-like programs in Java to solve combinatorial constraint satisfaction problems. A collection of generators oriented to world wide web applications is demonstrated on searches for HTML pages.

Keywords: continuation, generator, constraint satisfaction problem, combinatorial search, world wide web search, Java

1. INTRODUCTION

A continuation is a function (or procedure or method) used to represent a following computation. Continuations arose in denotational semantics as a way of defining the meaning of jumps and other sequencers in imperative programming languages. Here they are used in the object oriented programming language Java to implement combinatorial generators for constraint satisfaction problems and to write Prolog-like searches for world wide web pages.

The easiest languages in which to do continuation style programming are the lazy functional languages such as Haskell (Hudak *et al*, 1992). The applicative language Scheme (Hayes *et al*, 1986) even has continuations built into it as a predefined type. However, continuations can also be programmed in an imperative language such as C or Pascal if it is possible to pass a procedure as a parameter. The object oriented language Java does not allow procedures as parameters but references to *objects* can be passed as parameters and it is possible to define generator and continuation classes. These classes can be conveniently used in a Java application program or in an applet, if a task includes a search for the solution of a combinatorial problem, without having to switch to a non-deterministic language.

Continuations were first used to define jumps and other sequencers by Strachey and Wadsworth (1974) and Milne (1976). Strachey and Wadsworth attribute the origin of the idea to Mazurkiewicz's tail functions (Mazurkiewicz, 1971). Strachey, Wadsworth and Milne demonstrated that continuations could define arbitrary control mechanisms in programming languages. Their power has been used in semantics of Prolog (Nicholson and Foo, 1989; Finlay and Allison, 1993).

This paper demonstrates that continuations can be used in an object oriented programming language such as Java. It uses the technique to implement Java classes that can be used to write Prolog-like programs in Java to solve combinatorial constraint problems and, by also making use of

Manuscript received: August 1999
Associate Editor: Janet Smith

Java's internet package, to perform world wide web searches providing an alternative to adding web-awareness to a logic language as in LogicWeb (Loke *et al*, 1996).

2. CONTINUATIONS

This section describes continuations in the context of functional programming where they developed; it can be skipped if the idea is familiar. Section 3 describes continuations in Java.

A continuation is just a function *h* given to another function *f* to continue or to follow on from *f*. In some applications it can be thought of as *f*'s return-address plus suitable state information made explicitly available to *f:*

```
datatype Cont = u → v

fun f h x = h(g x)

f:Cont→(t→v)
h:Cont
g:t→u
x:t          {for some types t, u, v}
```

Note that the arrow, as in $u \rightarrow v$, constructs a function type which might be written as *function(u)v* in some imperative languages. The arrow is right associative so the parentheses in the type of *f* can be omitted. It is also common practice to omit parentheses in function applications, e.g. *g x* instead of *g(x)*, unless they are necessary to override operator priority. The function *f* is a *Curried* function, that is we write *f h x* rather than *f(h,x)*.

In the above artificial example, we think of *g* as the main body of *f*. Function *f* does some computation, *g*, on *x* and passes the result to *f*'s continuation, *h*. The formal parameter *h* represents whatever happens after *f*, hence the term continuation. Given an actual continuation *k*, *f k* is therefore a rather complex way of expressing the composition *k.g*. *F k* can also be read forwards as "do *f* and then do *k*" in contrast to *k.g* "do *g* and then do *k*". The full power of continuation programming comes from the fact that *f* can be reprogrammed to invoke its continuation *h* zero, once or several times as it chooses, possibly in a data-dependent way, e.g.

```
f h x = p( h(g  x), h(g' x))
```

where p is typically some simple computation that collates the results of h. In the following sections, logical disjunction is implemented by multiple uses of a continuation, the reason being that a successful search might come from one or another lines of computation.

In many uses of continuations the types *t* and *u* are the same and then:

```
datatype Cont = t → v

  f:Cont→t→v = Cont→Cont
  h:Cont
f h:Cont
```

In this case, *f* applied to a continuation yields a continuation, the types just fall out this way, which is convenient if *f* can be split into *f1* and *f2* as in *f h = f1(f2 h)*. This reads as "do f1 and then do (f2 and then do h)".

One particular use of continuations is in denotational semantics (Milne and Strachey, 1976) to define the semantics of imperative programming languages with jumps. The state of an imperative program is modelled by a type *State*. Each command, of type *Cmd*, in the imperative language performs a state transition. The environment, of type *Env*, holds information about variables etc. The final output or answer of a program is modelled by the type *Ans*. The meaning of commands is given

by a function *C* which takes the command being defined, an environment to map names onto bindings, a continuation being what (normally) happens after the command, and the current state:

```
datatype Cont = State → Ans

C:Cmd → Env → Cont → State → Ans
 =Cmd → Env → Cont → Cont

In particular
  C[cmd1; cmd2] e c s = C[cmd1] e (C[cmd2] e c) s
  C[goto L] e c s = e[L] s
so
  C[goto L; cmd2] e c s
= C[goto L] e (C[cmd2] e c)s
= e[L]s                      -- cmd2 ignored
```

The power of this form of *C* is that a jump to a label, as above, can discard the continuation that represents the normal sequential flow of execution and can instead execute another continuation which is bound to the label in the environment.

Note that a continuation is just a function used in a particular way, it does not need to be a special language mechanism.

Java does not allow procedures and functions, i.e. methods, to be passed as parameters of procedures and functions. It does however allow objects, instances of classes, to be passed as parameters. In what follows, generator and continuation objects are created and these have methods that behave in the same way as the continuations from denotational semantics.

3. CONTINUATIONS AND GENERATORS

The very term *object* in "object oriented programming" tends to imply notions of a rather concrete thing, probably with a state. On the other hand a pure function has no state. Functional programming might model a parser, for example, as a pure function from source language to parse tree. It might be useful or necessary to define internal states for the parser but each state would be a pure value and state transitions would be functions on these value.

Object oriented programming might model the parser as an object with state, having methods to query parts of the state and to advance the parsing process. It might alternatively concentrate on the result of parsing, i.e. the parse tree, and define a class of parse trees having the parser buried in its constructor.

Generating solutions to a combinatorial constraint satisfaction problem is closely related to parsing and indeed a grammar can be used to generate sentences of a language, for example to test a compiler or even for the purposes of humour (Bulhak, 1996).

Continuations can be used in functional and imperative programming (Allison, 1990) to solve combinatorial search problems. Typically there are constraints that a (partial) answer must satisfy; often one talks of partial solutions which are not yet complete but which satisfy the constraints so far and which may lead to one or more complete solutions. The particular problem will imply certain operators or *generators* that can be applied to extend or modify a partial solution. Basic atomic generators can be put together in various combinations using operators to form larger, more complex generators. Constraint satisfaction problems become difficult and require a search process in situations where a valid solution can be easily recognised but the order in which the basic generators need to be applied to produce such a solution is not easily determined. This situation can be modelled by the following functional types:

```
datatype Cont = State → State list

datatype Generator = Cont → State → State list
                   = Cont → Cont
```

A generator requires a continuation parameter to complete its branch of the search. At some point in a combinatorial search we have a partial solution or *State* and it may be possible to extend it into some complete, final solutions i.e. a *State list*, possibly empty. We model this in Java by providing interfaces that particular instances of *Generator* and continuations *GenContinuation* must implement:

```
public interface Generator
 { public void generate( GenContinuation c, State a );

public interface GenContinuation
 { public void continu( State a ); }
```

Note that *continue* is a reserved word in Java. Note also that we simply print solutions to the problem, rather than returning lists of solutions in a data structure. If necessary, the final continuation that recognises valid solutions could place them in a Java data structure.

One of the simplest generators adds an integer literal to a partial solution:

```
class GenInt implements Generator
 { private int n;

   public GenInt(int num) { n = num; }

   public void generate(GenContinuation c, State a)
    { c.continu( a.cons(n) ); } // prepend n to a
 }//GenInt
```

Cons is a method that prepends a value to a State; the word is taken from the programming language Lisp.

The logic programming language Prolog (Sterling and Shapiro, 1986) is based on first-order predicate logic. It is found to be powerful in artificial intelligence and deductive database applications. Predicate logic allows the high-level specification of solutions to many constraint satisfaction problems. Prolog also provides a non-deterministic search mechanism. This gives the inspiration for some Java generators below.

It is often necessary to apply the *conjunction* of two generators in sequence. To invoke generators *g1* and *g2* and then continuation *c*, invoke *gl* with the continuation formed from *g2* and *c:*

```
class GenAnd implements Generator
 { private Generator g1, g2;

   public GenAnd(Generator gen1, Generator gen2)
    { g1 = gen1; g2 = gen2; }

   public void generate(GenContinuation c, State a)
    { g1.generate(new GenCont(g2, c), a); }
 }//GenAnd

class GenCont implements GenContinuation
 { private Generator g; private GenContinuation c;
   public GenCont(Generator gen, GenContinuation con)
    { g=gen; c=con; }

   public void continu(State a) { g.generate(c, a); }
 }
```

The constructor *GenCont* takes a generator and a continuation and returns a continuation which invokes the given generator, passing it the given continuation as its parameter. If *g1.generate* fails, i.e. violates a constraint and is an unsuccessful search, it simply returns and does not invoke

GenCont(g2,c).generate. Similarly if *g1.generate* succeeds but *g2.generate* fails then *c.continu* will not be invoked.

GenAnd can be used *n* times by the Generator *GenDo* which forms the n-fold conjunction of a generator with itself:

```
class GenDo implements Generator
 { private Generator g;

   public GenDo( int n, Generator gen )
    { if( n <= 0 ) g = new GenSuccess();
      else if(n == 1) g = gen;
      else // n > 1
          g = new GenAnd(gen, new GenDo(n-1, gen));
    }

   public void generate(GenContinuation c, State a)
    { g.generate(c, a); }
 }//GenDo

class GenSuccess implements Generator
 { public void generate(GenContinuation c, State a)
    { c.continu(a); }  // succeeded ... so far!
 }
```

If there are zero generators to be composed by *GenDo* the result should obviously succeed, continuing with the rest of the computation and the trivial generator, *GenSuccess*, is included for this purpose.

To complement conjunction, the disjunction *GenOr* of two generators succeeds if either of them succeeds:

```
class GenOr implements Generator
 { private Generator g1, g2;

   public GenOr(Generator gen1, Generator gen2)
    { g1 = gen1; g2 = gen2; }

   public void generate(GenContinuation c, State a)
    { g1.generate(c, a);
      g2.generate(c, a);
    }
 }//GenOr
```

Note that the continuation *c* to *GenOr* is passed on twice, once to *g1.generate* and once to *g2.generate*. *GenOr* is used to define alternative branches of a search process. Naturally all solutions found by *g1* or by *g2* appear in the program's output.

Further generators can be defined such as *GenChoice* which is the n-fold disjunction of *GenInt;* it tries choices of integer literals between *1* and *n:*

```
class GenChoice implements Generator
 { private Generator g;

   public GenChoice(int n)  //  1 | 2 | ... | n
    { if( n <= 0 ) g = new GenFail();
      else if( n==1 ) g = new GenLiteral(1);
      else //  n > 1
        g=new GenOr(new GenChoice(n-1), new GenLiteral(n));
    }

   public void generate(GenContinuation c, State a)
    { g.generate(c, a); }
 }//GenChoice

class GenFail implements Generator
 { public void generate(GenContinuation c, State a)
    { } //do nothing, prune branch of search tree
 }
```

A choice between zero alternatives should obviously fail, i.e. not continue, and *GenFail* is included for this purpose.

As described earlier, a line of computation may lead to *zero* or more solutions. Some generators test constraints of a particular search problem and can be thought of as *filters*, that is they fail (return immediately) if their constraints are violated and otherwise succeed (continue). Failure prunes off a branch of the search tree.

4. N-QUEENS

The n-queens problem is to place n queens on an n×n chess board in such a way that no two queens threaten each other. It is a classic example of a constraint satisfaction problem, often used for illustrative purposes as it is both an interesting puzzle and easy to describe. Obviously, no two queens can be on the same row or column and therefore a solution can be represented by a permutation of the row-numbers amongst the columns. Not all permutations correspond to solutions of the n-queens problem however because the constraints on diagonals also have to be checked.

Solving the n-queens problem in a non-deterministic language (Floyd, 1967) is simplicity itself:

```
state := nil; // empty board
for 1 to n do // each queen
 { state := cons( choose(n), state );
   if state is not valid then fail
 }
```

Failure causes backtracking through other sequences of choices.

The classes defined in the previous section can now be used to solve the n-queens problem in Java. The central step is to choose a row for the queen on the current column and then to check that it does not threaten any previously placed queen:

```
new GenAnd(new GenChoice(n), // 1..n
           new Valid())
```

Recall that *GenChoice(n)* is the disjunction of *GenInt(1)* to *GenInt(n)*.

An n-fold conjunction *(GenDo)* of the central step will successfully place all n queens, if this is possible. First, the generator *nq* is constructed which will seek solutions:

```
// build the generator:
   Generator nq =
      new GenDo(n,
            new GenAnd(new GenChoice(n),
                       new Valid()));
// invoke it:
   nq.generate( new Success(), new State() );
```

When *nq* has been constructed, its *generate* method is invoked with a *Success* continuation and an initial, empty *State*. The resulting Java program is more verbose than the non-deterministic one but matches its structure closely.

For this problem the *State* class is implemented as a linked list which is easily programmed in Java. The final continuation class *Success* simply prints out the state. Details of checking the validity of an n-queens solution are omitted.

5. USING THE JAVA PROGRAM'S STATE

The previous implementation of generators and continuations in Java has the drawback that *State* is compiled into the definitions of the various generator classes. For the n-queens problem, and for several other combinatorial problems, a simple linked list is a suitable implementation of the state but most constraint satisfaction problems will require quite different definitions. Java classes cannot be parameterised on types or made generic so recompilation would be necessary if the state were changed.

One possible solution would be to make the state an interface specification or template which instances of state could implement but it is not at all clear what something as general as a "state" should specify in general over and above the most general Java class *Object*.

Another possibility is to use the state of the Java program (state as in imperative programming) to hold the components of the solutions to the problem. This involves dropping the *State* parameter of the *generate* and *continu* methods:

```
public interface Generator
 { public void generate( GenContinuation c ); }

public interface GenContinuation
 { public void continu(); }
```

Apart from deleting the *State* parameters, no change is required to the central infrastructure classes such as *GenAnd* and *GenOr.* However, those generators that manipulate the (partial) problem solution, such as *GenInt,* do require major changes because they need to communicate with each other. They previously used the *State* parameters of their generate methods to do this. Their constructors now need to be made aware of which variables in the Java state will be used for communication. For example, *GenInt.generate* needs to place an integer value in an integer variable:

```
class GenInt implements Generator
 { private IntVar iv; private int val;

   public GenInt(IntVar i, int n)
    { iv=i; val=n; }

   public void generate(GenContinuation c)
    { iv.set(val); c.continu(); }
 }
```

Unfortunately for *GenInt,* all Java parameters of basic types are passed by-value; there are no *reference parameters* in Java *in the usual sense of the term.* Therefore there is no way in which *GenInt.generate* can alter an *int* parameter given to it or to the constructor of *GenInt.* Consequently a mutable *IntVar* wrapper class having a *set* method must be used. Note, there is already a wrapper class, *Integer,* in the Java application programmer's interface (API) which is "useful when you need to pass int values by reference" (Flanagan, 1996), but its value cannot be changed! There does not seem to be any particular reason why Java could not have either by-reference or by-input-output parameters although it might be better to call them "var" parameters, after Pascal, to avoid confusion with other uses of the term "reference" in Java. The deletion of the address and pointer operators, compared to C, was done on sound security grounds but does have the side-effect of restricting the ways in which Java routines can output results.

Note that it is *GenInt's generate* method not its constructor that sets the *IntVar*'s value. This has to be the case because a particular *GenInt* may be invoked many times and one *IntVar* may be used for many purposes in a particular program.

With these new generators, the n-queens program becomes:

```
IntVar [] rows = new IntVar[n];  // create nxn board
for(int i=0; i < n; i++) rows[i] = new IntVar(0);

Generator nq = new GenSuccess();

for(int i=n-1; i >= 0; i--)
   nq = new GenAnd(
            new GenAnd(new GenChoice(rows[i], n),
                       new Valid(rows, i)),
            nq );

nq.generate( new QueenSuccess(rows) );
```

The central step is still to choose a row number for a new queen and check that the constraints are not violated. Now *IntVar rows[i]* is used to hold the row number and *Valid* is told to check that the i^{th} queen does not threaten any previously placed queens. *Nq* is a conjunction of *n* copies of this central step. The *QueenSuccess* continuation prints out a solution. The n-queens code above plus *Valid* and *QueenSuccess* are the only parts of the program that are unique to the n-queens problem. This n-queens program is included as an appendix.

6. WORLD WIDE WEB SEARCHES

The main interest in Java is that it very suitable for internet applications, particularly those involving the world wide web, and that it comes with a large application programmer's interface (API) containing many routines for that purpose. Java *applets* can be embedded in hypertext pages. Java programs are compiled into byte code and interpreters are widely available for many computers and are built into most new web browsers.

The world wide web allows the publishing of *pages* written in hypertext markup language (HTML). Such a page (or other material) can be found by its universal resource locator (URL) or internet address. An HTML page can contain hyperlinks, i.e. URLs, pointing to other HTML pages and other types of material.

It is possible to program a collection of generators to assist in world wide web queries. For example, *GenURLsFromURL* generates the URLs found in the page specified by a given URL, and *GenURLcontainsString* succeeds if a page contains a given string and fails otherwise. Using these generators a page *url2* can be defined as "interesting" if it contains the string "biolog" (as in biology, biological, biologist, ...) or if it contains a hyper-link to a page *url3* which contains the string "tRNA" (as in transfer RNA). The search starts from *url1* which refers to a query for "dynamic programming" through the AltaVista search engine (1996):

```
RefObj url1, url2, url3;

url1.set(new URL("http://altavista...cgi-bin/query?" +
                 "...&q=dynamic+programming")

Generator interesting =
 GenOr(new GenURLcontainsString(url2, "biolog"),
       new GenAnd(new GenURLsFromURL(url2, url3),
                  new GenURLcontainsString(url3,"tRNA")))

Generator gu =
 new GenAnd(new GenURLsFromURL(url1, url2),
            interesting );

gu.generate(new PrintURL(url2));
```

The class *RefObj* is a general mutable reference to an arbitrary object and generators can output information through RefObj parameters such as *url2* and *url3*.

The reader might quibble about the definition of interesting above but this is just an example! It might have been expressed as follows in some hypothetical and suitably augmented version of Prolog:

```
interesting(URL2) :- genURLcontainsString(URL2,"biolog").

interesting(URL2) :- genURLsFromURL(URL2,URL3),
                     genURLcontainsString(URL3, "tRNA").

?- url("http://altavista...cgi-bin/query?" +
       "...&q=dynamic+programming", URL1),
   genURLsFromURL(URL1,URL2), interesting(URL2).
```

The Java version is more verbose due to the restrictions of Java syntax but there is a clear correspondence between the two versions. The programmer of the Java version can also make full use of the Java language, the Java program's state and the Java API to program new generators and to do further processing that might require imperative-style algorithms and data-structures on the grounds of efficiency or of expressiveness.

7. CONCLUSION

Continuations and generators have been programmed in the object oriented programming language Java. The technique provides a non-deterministic control mechanism which allows combinatorial searches to be quickly programmed. Experiments with a set of generators oriented towards world wide web applications show that sophisticated internet searches can be expressed in this style.

The use of predefined generators is akin to writing Prolog-like or predicate-logic based queries and specifications, although it is somewhat more verbose due to the restrictions of Java syntax. On the other hand, all the facilities of the imperative language Java, its state and its API are available when a pure non-deterministic search is inefficient or inadequate.

Footnote: Since this paper was first written, Java 1.1 (Flanagan, 1997) has been released; it does not involve any changes to the underlying Java virtual machine. Amongst other features, Java 1.1 contains *inner classes* and *anonymous classes* which would allow some of the Java code presented in this paper to be shortened slightly. Various *proposals* also exist to extend the Java class and type system, e.g. with polymorphism as in Pizza (Odersky and Wadler, 1997), which would facilitate the use of techniques described here.

8. REFERENCES

ALLISON, L. (1990): Continuations implement generators and streams. *Comp. Jrnl.* 33(5): 460-465.

AltaVista Search Engine (1996): Digital Equipment Corporation.
URL: http://www.altavista.com/ was http://altavista.digital.com/

BULHAK, A. (1996): On the simulation of postmodernism and mental debility using recursive transition networks. Dept. of Computer Science, Monash Univ. TR 96/264.
URL:http://www.cs.monash.edu.au/cgi-bin/postmodern

FINLAY, A. and ALLISON, L. (1993): A correction to the denotational semantics for Prolog of Nicholson and Foo. *ACM Trans. Prog. Lang. & Sys.* 15(1) January: 206-208.

FLANAGAN, D. (1996, 1997): *Java in a Nutshell.* O'Reilly & Assoc. 2nd edition, Java 1.1: 1997.

FLOYD, R.W. (1967): Nondeterministic Algorithms. *Jrnl. ACM* 14(2): 636-644.

HAYNES, C.T., FRIEDMANN, D.P. and WAND, M. (1986): Obtaining coroutines with continuations. *Comp. Languages* 11(3/4).

HUDAK, P., PEYTON JONES, S., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J., GUZMAN, M.M., HAMMOND, K., HUGHES, J., JOHNSSON, T., KIEBURTZ, D., NIKHIL, R., PARTIAN, W. and PETERSON, J. (1992): Report on the Programming Language Haskell, version 1.2. *Sigplan* 27(5) May.

LOKE, S.W., DAVISON, A. and STERLING, L. (1996): CIFI: An intelligent agent for citation finding on the world wide web. Technical Report 96/4, Dept. Comp. Sci. Melbourne Univ.

MAZURKIEWICZ, A.W. (1971): Proving algorithms by tail functions. *Inf. & Control* 18: 220-226.

MILNE, R. and STRACHEY, C. (1976): *A Theory of Programming Language Semantics* (2 vols). Chapman & Hall.
NICHOLSON, T. and FOO, N. (1989): A denotational semantics for Prolog. *ACM Trans. Prog. Lang. & Sys.* 11(4) October: 650-665.
ODERSKY, M. and WADLER P. (1997): Pizza into Java: Translating theory into practice. Proc. 24th ACM Symp. on Principles of Programming Languages (POPL) January.
STERLING, L. and SHAPIRO, E. (1986): *The Art of Prolog.* MIT Press.
STRACHEY, C. and WADSWORTH, C.P. (1974): Continuations: a mathematical semantics for handling full jumps. PRG-11 Oxford University.

APPENDIX: N-QUEENS PROGRAM OF SECTION 5

```
// Place n Queens on an nxn chess board so that no two Queens threaten each
// other. There must be exactly one Queen per row, so it is sufficient to
// know the row number of the Queen on each column.

public class Queens
 { public static void main(String[] args)
    { int n = Integer.parseInt(args[0]);  // command line:- how many Queens?

      IntVar [] rows = new IntVar[n];                    // create nxn board
      for(int i=0; i < n; i++) rows[i] = new IntVar(0);

// build the n-Queens solver:
      Generator nq = new GenSuccess();

      for(int i=n-1; i >= 0; i--)                        // for i=1..n
         nq = new GenAnd(
                  new GenAnd(new GenInts(rows[i], n),    //    try
                             new Valid(rows, i)),        //    test
                  nq );

// now run it:
      nq.generate( new QueenSuccess(rows) );
    }
 }//Queens


class Valid implements Generator // Filter out violations
 { private IntVar rows[]; private int col;

   public Valid(IntVar[] r, int c) { rows = r; col = c; }

   public void generate(GenContinuation c)
    { boolean clash = false; int rC = rows[col].value();
      for(int i=0; i < col; i++)        // test for queen threats
       { int rI = rows[i].value();
         if( rI == rC || rI == rC-col+i || rI == rC+col-i ) clash = true;
       }
      if( ! clash ) c.continu(); // else fail, i.e. prune search tree
    }
 }//Valid


class QueenSuccess implements GenContinuation // process a solution
 { private IntVar[] rows;

   QueenSuccess(IntVar [] r) { rows = r; }

   public void continu() // print the solution
    { for(int i=0; i < rows.length; i++)
         System.out.print(rows[i].value()+" ");
      System.out.println();
    }
 }//QueenSuccess
```

BIOGRAPHICAL NOTES

Lloyd Allison is a reader in the School of Computer Science and Software Engineering at Monash University.

He obtained a PhD in Computer Science at Manchester University in 1976 and has worked at Melbourne University, the University of Western Australia, and now Monash.

His main professional interests are in algorithms and data structures, programming languages, weird and otherwise, inductive inference and computing for molecular biology.