Continuations Implement Generators and Streams

L. ALLISON
Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia

Continuations are used to program generators and a variation on stream functions. The generators allow backtracking or non-deterministic search. The streams process sequences of values in stages without the creation of intermediate data structures (lists). Both are programmed in a functional language without special extensions. This brings two useful problem solving models into pure functional programming.

Received February 1988, revised July 1988

1. INTRODUCTION

A continuation is a function used to represent a following computation. Here, continuations are used to implement generators and a variation on streams. The aim is to explore the continuation style of programming while bringing two important problem solving models into pure functional programming. Generators enable compact backtracking or non-deterministic programs to be written. They are a central feature of some symbolic processing languages such as Icon[5] and greatly simplify the writing of combinatorial programs. A stream processes a sequence of values. Conventionally a sequence is represented by a list; it is often natural to process a sequence in stages and this can lead to the creation of intermediate lists which are eventually garbage collected. The streams in this paper are collections of functions. A function produces values one at a time which are processed and eventually consumed by other functions. At no stage does an intermediate list exist, in this sense the stream functions are listless.[15, 16] Often the collection of functions can, in principle, be implemented as a set of coroutines or as processes and can operate in small or even fixed space.

The language of choice for implementing generators and streams by continuations is a lazy functional language. For both generators and streams a set of basic operators is defined. Some operators only achieve full generality in a lazy language but all can be used in a strict functional language. Most operators can even be used with benefit in an imperative language. The definition of some operators is difficult but it is only done once; their use is simpler and leads to compact and readable programs. The n–queens problem is used to illustrate generators and the sieve of Eratosthenese to illustrate streams. In previous work,[1] continuations were used to parse non-deterministic grammars and to merge two sort trees.

A pure functional language is all that is required to use the techniques described in this paper. In contrast, at least one language, Scheme,[6] supports a special type, also called continuations, as first class values. A (Scheme) continuation is passed to a function by the call/cc operation or call with current continuation:

```
e.g. f(call/cc g)
        where fun g k = ...k(h)...
```

(Scheme uses different syntax.) Scheme is a dialect of Lisp that uses applicative-order evaluation (call by value) and has side-effects via set! In the example above g is called and k is bound to g's continuation. K appears to be a function but if it is called control returns apparently from the call/cc with value h; g is not resumed later unless special steps are taken. Scheme continuations have equivalent power to regular continuations and Haynes *et al.*[6] use them to define coroutines. The continuations used in this paper are ordinary, pure functions used in a particular way.

Continuations arose in denotational semantics to define the meaning of sequencers. They were first used in this way by Strachey and Wadsworth[14] and by Milne.[10] Strachey and Wadsworth attribute the origin of the idea to Mazurkiewicz's tail functions[9] which were used in the proof of programs. Strachey, Wadsworth and Milne demonstrated that continuations could define arbitrary control mechanisms in programming languages; the aim here is to use continuations in regular programming tasks.

1.1. Continuations

A continuation is a function *h* given to another function *f* to continue or to follow on from f:

```
datatype Cont = u→v   {→ denotes a function type}
fun f h x = h(g x)    {define f}
f :Cont→t→v           {types of f, h, g, x}
    h :Cont
    g :t→u
    x :t              {for some types t, u, v}
```

In this particular example, 'fh' is a rather complex way of expressing h ∘ g. Informally, we call g the body of f and can read 'fh' as 'do the body of f and then do h' or just as 'do f and then do h'. Note that the use of continuations is strictly more powerful than composition as f can be defined so as not to call h at all or to call it several times. This versatility is used in the applications of continuations in the following sections.

In many cases the types t and u are equal and then

```
datatype Cont = t→v

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

Note that a continuation is just a function. No new language mechanism is needed for its use.

2. GENERATORS

A non-deterministic program may generate many answers. A non-deterministic *computation* is a function from partial answers to lists of answers. To keep faith with denotational semantics, such a computation is called a *continuation* or a *Cont*:

```
datatype Cont = Answer → Answer list
```

A generator processes a (partial) answer in some way, for example by extending it, modifying it or testing it; the new answer may then be passed on to the following computation. Formally, a generator takes a continuation and produces a continuation; equivalently it takes a continuation and an answer and produces a list of answers:

```
datatype Generator = Cont → Cont
                   = Cont → Answer → Answer list
```

Many symbolic processing languages use generators. Icon[5] possesses explicit generators. The non-deterministic generation of solutions is a feature of Prolog. Continuations are present in denotational semantics of Prolog[8,12] and in the implementation of Prolog in functional programming.[3] The use of continuations is also implicit in the organisation of Prolog's trail stack.

2.1. Standard operators

Standard generators and operators on generators are defined in this section to form a basic building kit. Where possible familiar and meaningful names are chosen – such as pipe to suggest a form of composition.

In many combinatorial problems, an answer is a list of some kind of value:

```
datatype Answer = t list   {for some type t}
```

Often an answer is built up element by element. A simple example of a generator is *literal* which prepends a constant to a partial answer:

```
fun literal c h a = h(c::a)
    literal:t → Generator
           = t → Cont → Cont
    c :t
    h :Cont
    a :Answer
```

Note that '::' is the infix list constructor adopted from ML. The extended answer c::a is passed to the continuation h.

Some convenient operators on generators can be defined. *Pipe* connects two generators together in sequence:

```
fun pipe g1 g2 h a = g1(g2 h) a
    pipe :Generator → Generator → Generator
    g1, g2 :Generator
```

Reading informally, to do pipe g1 g2 and then h do g1 and then do g2 h. Any answers generated by g1, given a, are passed to g2 h. Note that simple functional composition, ∘, cannot be used because g1 may choose not to call its continuation or to call it several times.

Sometimes it is necessary to connect several copies of a generator together in sequence:

```
fun do n g =
      if n = 0 then success
      else pipe (do (n−1) g) g
   do :Int → Generator → Generator

fun success h a = h a
   success :Generator
```

Success is a restricted identity function introduced only for the sake of its name.

The operator *either* takes two generators and produces a generator which behaves as the non-deterministic choice between them:

```
fun either g1 g2 h a =
      append (g1 h a) (g2 h a)
   either :Generator → Generator → Generator
```

Note that either passes h to g1 and to g2 so that h may be called multiple times or not at all. Any answers that g1 h and g2 h produce are collected together. This implements left-to-right depth-first search. If implemented in a strict language, either loops if g1 loops or if g2 loops. If implemented in a lazy language, either produces infinitely many solutions if g1 does or if g2 does.

It is often necessary to generate a *choice* from the integers {1..n}:

```
fun choice n =
      if n = 0 then fail
      else either (literal n) (choice (n−1))
   choice :Int → Generator

fun fail h a = [ ]
   fail :Generator
```

Choice puts n, n − 1 and so on down to 1 on the front of *each* answer that it is given. Fail simply discards its continuation and returns no solution which is represented by the empty list. The non-deterministic operators choice and fail were first proposed by Floyd.[4]

In order to run generators, it is convenient to define a final continuation *fin* and a *run* operator:

```
fun fin a = [a]
   fin :Cont

fun run g = g fin [ ]            {excuse pun}
   run :Generator → Answer list
```

Using the operators defined above it is possible to write some simple programs. The program

```
run( do 3 (choice 2) )
```

produces all sequences of length 3 over the alphabet {1,2}.

Many combinatorial backtracking programs involve the generation and testing of answers. It is possible to *filter* (partial) answers according to a predicate:

```
fun filter p h a =
      if p a then h a
      else [ ]
   filter :(Answer → Bool) → Generator
```

If an answer, a, fails the test, p, the empty list of solutions is returned. If it passes the test it is given to the continuation h.

2.2. An example: n–Queens

Using the operators of the previous section, the n–queens problem can be programmed. First, an auxiliary function, valid, is needed to check a partial solution. A solution is represented by a list of row numbers. Valid checks that the last queen added to a solution does not threaten any other queen. (Valid is a conventional predicate and does not employ continuations.)

```
{ assuming datatype Answer = int list }

fun valid nil = true |
    valid h::t = v 1 t
    where fun
      v row nil = true |
      v row x::y =
        if h = x or h+row = x or h−row = x
          then false
          else v (row+1) y
```

We are now in a position to write the n–queens program proper. An informal solution to the n–queens problem might be given as follows: do the following n times, choose a row on which to place the queen in the current column and then check that it is consistent with the previous choices made so far. Continue until all n queens have been placed. This solution is directly implemented in generators by continuations:

```
run (do n place_one_queen)
  where fun
    place_one_queen = (pipe (choice n) (filter
                         valid))
```

It resembles Floyd's n–queens program,[4] which translates into Pascal-like notation as:

```
for col:= 1 to n do
begin row[col]:= choice (n);
                          {non-deterministic choice}
      if not valid(row[1..col]) then fail
                                        {backtrack}
end;
{success}
```

Although Floyd's program was given in an imperative, non-deterministic language, the variables are each assigned once only. (Arguably this is true even of col if one considers the **for** loop to be a **forall** construct.) This might have lead one to suspect the existence of the short functional program given above.

2.3. Imperative languages

It is possible to do some programming with continuations in an imperative language. This brings some of the benefits of functional programming to the imperative programmer. The biggest inconvenience, and that is all it is, in a stack-based language is the inability to return a (local) function result and the lack of curried routines. However the function types of the previous section can be uncurried and programmed, even in Pascal.

```
{ choose:Int→Cont→Cont = Int→Generator }
procedure choose( n:integer;
                  procedure cont (L:list);
                  L:list);
  var i:integer;
begin
  for i:= 1 to n do
    cont( cons(i, L) )
end;

{ filter:predicate−> Generator }
procedure filter(function p(L:list):boolean;
                 procedure cont(L:list);
                 L:list);
begin if p(L) then cont(L) {else fail}
end;

{ doo:Int→Generator→Generator }
procedure doo(n:integer;
              procedure gen ( procedure cont
                               (L:list); L:list);
              procedure cont(L:list);
              l:list);
  procedure gencont(L:list);
  begin gen(cont, L) end;
begin
  if n = 0 then cont(L)
  else doo(n−1, gen, gencont, L)
end;

function valid(L:list):boolean;
  ... tests if partial solution L valid ...

procedure queen(n:integer);
  procedure choosevalid(procedure cont(L:list);
                        L:list);
    procedure validcont(L:list);
    begin
      filter(valid, cont, L)
    end;
  begin
    choose(n, validcont, L)
  end;
begin
  doo(n, choosevalid, success, nil)
end;
```

The executable commands are compact; they form a functional program coded in Pascal. The declarations make the Pascal version longer than the functional version but are added almost mechanically. Lacking curried routines, it is necessary to introduce some auxiliary routines such as choosevalid. Rather than return a list of solutions, it is convenient to have the solutions printed in the Pascal output file. This leads to some minor programming differences, for example filter does not return nil if the predicate fails, it simply does not call its continuation.

3. STREAMS

The processing of a sequence of values in small stages is a common occurrence. If a sequence is represented as a list, intermediate lists are created and garbage collected later. Burge[2] describes an alternative to lists called streams which allow some infinite lists to be computed. To Burge, a stream is a function which produces the first value of a sequence and another stream; the latter represents the rest of the sequence. In this section another variation is described which uses *source*, *agent* and *sink* functions, collectively called streams. A source is a function which produces a first value, but it has a parameter, a sink, which uses that value. A sink is a function which consumes a value and has a parameter, a source, which produces some more values:

```
datatype source =       sink   → int list
and      sink   = int → source → int list
```

A source has a sink as a continuation and a sink has a source as a continuation.

Agents perform intermediate steps and are defined in the following section as are various standard stream functions.

3.1. Standard operators

A simple example of a source produces a sequence of ascending integers:

```
fun range lo hi outp =
        if lo <= hi
          then outp lo (range (lo+1) hi)
          else nil
    range :int → int → source
    outp  :sink
```

The first value, lo, is passed to the sink, outp. The source (range (lo+1) hi) is also passed to outp which will probably, but not necessarily, call it to produce lo+1 and so on up to hi.

The null source produces no values:

```
fun null_source outp = nil;
    null_source :source
```

The simplest useful sink records its input:

```
fun print n ip = n::(ip print)
    print :sink
    n     :int
    ip    :source
```

The input, n, is put into the output list and the source, ip, is invoked for more elements of the sequence.

The black_hole discards its input values:

```
fun black_hole n ip =
        ip black_hole;    {alias /dev/null}
    black_hole :sink
```

The black_hole does not return nil but calls the input source ip because other functions may cause some output later.

A source and a sink can be combined to produce a list of values:

```
range 1 10 print   { = [1,2,3,4,5,6,7,8,9,10] }
```

The source, range 1 10, and the sink, print, act as coroutines. Range produces an integer and passes it to print together with itself as parameter. Print adds the integer to the output and resumes range and so on. The program can be read as 'generate integers in the range 1 to 10 and then print them'.

As well as sources and sinks there are intermediaries or *agents*:

```
datatype agent = sink → int → source → int list
               = sink → sink
```

An agent processes some input and may pass it on. An agent can use its sink parameter or its source parameter as the continuation to be called next. Note that not only does agent = sink → sink but that, except for the order of parameters, agent ≈ int → source → source:

```
given a:agent, n:int, ip:source then
(λ outp. a outp n ip) :sink → int  list = source
```

As examples, the agents *even* and *odd* pass on half of their input. Odd passes on the first value, the third value and so on. Even passes on the second value, the fourth value and so on.

```
fun even outp n ip =
        ip (odd outp)
and odd outp n ip =
        outp n (λ outp′. ip (even outp′))
    even, odd :agent
```

Even discards the first value, n, and asks for a second value from the input source ip whereas odd outputs the value immediately.

It is convenient to have an operator *run* to link a source, an agent and a sink together:

```
fun run ip mid outp = ip (mid outp)
    run :source → agent → sink → int list
     = source → agent → source

run (range 1 10) odd print { = [1,3,5,7,9] }
```

Run overrides the default binding of application with brackets () and its use improves readability in some circumstances.

The operator *pipe* connects two agents together:

```
funpipe a b outp n ip =
       a (b outp) n ip
    pipe :agent → agent → agent

run (range 1 10) (pipe odd even) print { = [3,7] }
```

As was the case when connecting two generators, the agents in the pipe, a and b, can behave more generally than their composition.

A sequence can be *filter*ed by an agent according to a predicate:

```
funfilter p outp n ip =
       if p n
         then outp n (run ip (filter p))
         else run ip (filter p) outp
    filter:(int → bool) → agent
```

If the value n passes the test p it is passed to the sink outp. If n fails the test the source ip is called for the next value. It can be seen that no list data structure is built to contain the reduced sequence of values.

3.2. An example: Sieve of Eratosthenese

Once the building blocks of the previous section are defined they can be used to write compact stream processing programs. For example, a sieve of Eratosthenese can be programmed in this style:

```
funsieve outp n ip =
        outp n (run ip (pipe (remove_multiples n)
                sieve))
    sieve:agent
where fun
  remove_multiples n = filter (not∘(mult n))
    where fun
      mult a b = (b mod a) = 0
```

```
run (range 2 20) sieve print
                        { = [2,3,5,7,11,13,17,19) }
```

Reading from left to right, the sieve is similar to its English description. Sieve is given an output mechanism outp, an integer n and an input mechanism ip. It first outputs the integer (outp n) and then does some more input (run ip). The results of this input have multiples of n removed and are sieved again.

3.3 More operators

It is possible to imagine and to program a great many other operators on sources, agents and sinks. As a few examples consider the following:

The infix operator on lists, '?', is defined by

```
nil ? L = L   |
(h::t) ? L = h::t

funzip (ip1,ip2) outp =
     ip1 (λ m,ip1'.outp m (zip (ip2,ip1')) )
   {alternates values from ip1 and ip2}
   zip :source × source → source

funmerge (ip1,ip2) outp =
     ip1 (λ m,ip1'.ip2(λ n,ip2'.switch m ip1' n ip2'
        outp)
                  ? outp m ip1')
   ? ip2 outp
where fun switch m ip1 n ip2 outp =
  if m <= n
    then outp m (λ outp'.ip1 (λ m',ip1'.switch m'
                  ip1' n ip2 outp')
                           ? outp' n ip2)
    else switch n ip2 m ip1 outp

merge :source × source → source

funT ip = (ip,ip)
   T :source → source × source

funsplit ip =
     ( (run ip odd), (run ip even) )
   split :source → source × source
```

Zip alternates from two sources to produce a new source. It first calls ip1, passing it a sink to use the value, m, produced by ip1 and then zips ip2 and ip1. Split is zip's inverse. Merge merges the values, which are assumed to be ascending, from two sources into one source. It is complicated by the need to get the initial values, m and n, produced by each source and by the need to copy the tail of the last sequence when the first runs out. T makes two copies of a source; it is named after a T-joint in plumbing.

3.4. Implementation

There is a small, seemingly unavoidable, asymmetry between sources and sinks. In the coding given, sources take the initiative in producting values which are 'pushed' into the sinks – the programs are data driven. There is an equivalent output-driven coding where the sinks demand the values.

The sources, sinks and agents described in the previous sections have been programmed in standard ML.[11] In that language it is necessary to add type constructors to the recursive datatypes source and sink; otherwise the programs are as above. Algol-68 could be used if the functions were uncurried although more local function definitions would be necessary. Pascal cannot be used as it does not support recursive function types.

Some of the simpler stream programs can be executed in a fixed amount of space, consider:

```
range 1 10 print
```

This program needs only two environments. Range calls a function as its last action – it is tail-recursive.[13] Range's environment can be overwritten provided that there is no other reference to it. Print is tail-recursive mod cons; it can reserve a list cell, fill in the head of the cell and then, as its final action, call a function to compute the tail of the cell. Equivalently, range and print can act as coroutines. Some functional language implementations that are based on combinators, and so do not use environments, exhibit correct tail-recursive behaviour.[7]

The primes program cannot be executed in fixed space but it can be viewed as a number of filtering processes each of which can be executed in a fixed amount of storage. No intermediate lists are needed for passing values between sieves.

4. CONCLUSIONS

Continuations allow the programmer to specify the flow of control of a program (this is why they are useful in denotational semantics). They allow backtracking programs such as the generators to be written. They also allow the stream processing functions – sources, sinks and agents – to pass control between each other as coroutines or as processes. At first sight functional languages seem to be poor in control mechanisms, possessing only **if**, application, composition and recursion. In reality this simplicity is richness and continuations are one way of getting a desired control regime.

Generators and stream processing are important models in solving certain types of problem. Continuations bring them into pure functional programming without the need for new language mechanisms.

Continuations involve a novel style of programming but the programs that result (n–Queens, primes) are often close to their English descriptions. They have something of an imperative flavour – do this and then do that – but they are functional and free of side effects. The programs can be effectively implemented by optimisations such as tail-recursion.

REFERENCES

1. L. Allison, Some applications of continuations. *The Computer Journal* **31** (1), 9–11 (1988).
2. W. Burge, *Recursive Programming Techniques*. Addison Wesley (1975).
3. M. Carlsson, On implementing Prolog in functional programming. *International Symposium on Logic Programming*, pp. 154–159 (1984).
4. R. W. Floyd, Nondeterministic algorithms. *Journal of the ACM* **14** (2), 636–644 (1967).

5. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*. Prentice Hall (1983).
6. C. T. Haynes, D. P. Friedmann and M. Wand, Obtaining coroutines with continuations. *Computer Languages* **11** (3/4), 143–153 (1986).
7. T. Johnsson, *Efficient compilation of lazy evaluation*. Programming Methodology Group, University of Goteborg and Chalmers University of Technology, TR40 (Feb. 1987).
8. N. D. Jones and A. Mycroft, Denotational semantics of Prolog. *International Symposium on Logic Programming*, pp. 281–288 (1984).
9. A. W. Mazurkiewicz, Proving algorithms by tail functions. *Information and Control* **18**, 220–226 (1971).
10. R. Milne and C. Strachey, *A Theory of Programming Language Semantics*. (2 vols). Chapman & Hall (1976).
11. R. Milner, *The standard ML core language*. Department of Computer Science, University of Edinburgh (1986).
12. T. Nicholson and N. Foo, *A denotational semantics for Prolog*. Basser Department of Computer Science, University of Sydney (1985).
13. G. L. Steele, Debunking the expensive procedure call myth. *Proc. Annual Conference ACM*, pp. 153–162 (1977).
14. C. Strachey and C. P. Wadsworth, *Continuations: a mathematical semantics for handling full jumps*. PRG-11, Oxford University (1974).
15. P. L. Wadler, *Listlessness is better than laziness*. CMU-CS-85-171 Computer Science Department, Carnegie-Mellon University (Aug. 1984).
16. P. Wadler, Listlessness is better than laziness II: composing listless functions. *Proc. Workshop on Programs as Data Objects*. Lecture Notes in Computer Science, vol. 217. Springer Verlag (1985).

Announcement

10–13 JUNE 1991

Eindhoven, The Netherlands, Conference on Parallel Architectures and Languages, Europe Call for Papers

Conference objectives

The PARLE '91 Conference will follow the tradition of the previous two conferences, PARLE '87 and PARLE '89, in being organised as a wide and representative international meeting of researchers in the fields of theory, design and application of parallel computing.

The conference programme includes invited presentations and contributed papers on current research.

The conference initiative is taken by project 2427 (TROPICS) of the European Strategic Programme for Research and Development in Information Technology (ESPRIT) of the Commission of the European Communities.

Submission of papers

Participants who wish to present papers are invited to send 5 copies of a full draft paper not exceeding 6000 words to the official conference mailing address before 15 October 1990. The conference language is English. Acceptance of papers will be notified to the authors by 1 February 1991. Camera-ready copies of accepted papers in final form are due 15 March 1991. Accepted papers will be published by Springer Verlag in the series *Lecture Notes in Computer Science*, and will be available at the conference.

Conference scope

Parallel architectures and systems
- formal modelling
- specification and verification
- data flow, inference and reduction machines
- interconnection networks
- multiprocessor design issues (VLSI, WSI, RISC)

Parallel programming
- formal programming methodologies
- specification and verification
- constraint-based concurrent programming

Parallel languages
- parallel programming primitives
- semantics for parallelism
- communication protocols
- implementation models

Parallel algorithms and complexity
- design and analysis of parallel algorithms
- complexity of parallel computations
- mappings of parallel algorithms

Applications of parallelism
- systolic arrays and regular computations
- neural networks and computing
- parallel and distributed databases
- parallelism in symbolic computing

Conference location and accommodation

The PARLE '91 Conference will be held at the Congress and Meeting Centre 'De Koningshof'. Mailing address: P.O. Box 140, 5500 AC Veldhoven, The Netherlands. Tel: +31 40 537475. Telefax: +31 40 545515.

The full fee for the conference is around Dfl 1000.

Official Conference mailing address

Mr F. Stoots, Philips Research Laboratories, P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands. Fax: +31 40 744758; E-mail: stoots@tropicsa.prl.philips.nl.