Implementation of Generators Using Continuations

Anthony Tsui

Table of Contents

Introduction

The implementation of generators using continuations is focused on two different concepts in programming, continuations and generators. Continuations are best understood as a type of abstract representation of the control state in a program. While it is well understood that when a function with a computation is called, the resulting computation is expected to return to the function. Continuations are a method of handling this return and choosing where in the program to progress to with the resulting computation. This is done with a concept called escape procedures that is strongly connected with continuations, which we will explore more deeply later. For this section, all that needs to be understood is that continuations allow us to move to arbitrary parts of a program with a computation or value from a different function or procedure.

When discussing generators, it may be helpful to first think about iterators. In a typical for loop in other programming languages, a for loop is utilized to perform computations on each element as it iterates through. The loop essentially steps through the elements in the structure, pausing at each element and processing it before moving on. A generator is similar to this respect in that when called, the generator will return an element of its respective structure and then suspend itself. When called again, it will return the next element in the list and suspend itself again. We can now see the correlation between continuations and generators, in order to properly create a generator we will need to utilize continuations to save our states and escape the loop.

Brief History of Generators & Continuations

While generators themselves may be a rarer routine compared to its similar functioning object, iterators. Generators were first seen in 1975 being utilized in CLU and was initially utilized for string manipulation. However, it is more practical delve into the history of iterators rather than generators, as they serve very similar functionalities and all generators are iterators. Iterators, and by extension generators, is an object utilized to traverse a container or data structure in programming. While the functionality of an iterator has not changed since its initial days, the representation of iterators/generators has changed from language to language. In more modern programming languages, iterators and generators are usually relegated to for loops, while loops, and explicit iterators. While some languages still come with and utilize built-in generators, for loops and iterators are far more common.

Continuations were first utilized decades ago in 1960, in order to control the processing of procedures and jumping in and out of blocks. While not explicitly stated as continuations, the ideas behind a stack, control, environment, and dump was the precursor to the creation of continuations. Moving forward to more modern times, continuations have taken more recognizable forms in escape functions, catch and exception handling, and of course, call-with-current-continuation or call/cc.

Significance of Generators & Continuations

The significance of generators was touched upon during the previous historical overview. For many programmers, for loops and iterators are commonly used in order to access the elements of a container or structure such as a tree, list, or array. As mentioned, the purpose of a for loop is to access each element inside of a container and perform computations upon it, doing so for each element in the targeted container. While generators are very similar in that they can provide the elements inside of a structure, but have the benefit of suspending the iteration until called upon again. This is an appropriate time to mention that while iterators are objects, generators are functions that behave like the iterator object. This property of suspending iteration is notably important, as compared to for loops and iterators, we can choose to stop at a certain element if we decide to. Of course, modern languages support similarly functionalities with statements such as break in python in order to exit the loop.

Likewise, the importance and effectiveness of continuations are just as intuitive as iterators/generators. Modern control stacks in contemporary languages are automatically determined by the language itself, handling errors with exceptions or determining what order to execute functions. Exception handling is often built in to the language being utilized, but is a prime example of continuation and control state manipulation. Continuations are a more abstract and general purpose control state, allowing more explicit manipulation of the control stack by the programmer. There are many usages for continuations, with the two we are mainly interested in being saving

states and escape procedures.

Escape procedures are procedures utilized to escape from a function or procedure. The usages of escape procedures can come to mind rather quickly, for example in a manner similar to if or while statements. If we are given a list and target element to find within said list, we may call an escape procedure once this element is found to exit the iteration and escape to a different procedure with our element, such as to return true to signify we have found said element. Observe the code snippet below:

```
(+ (call/cc
    (lambda (toEscape)
      (* (toEscape 3) 4)))
   3)
```

```
Language: R5RS; memory limit: 128 MB.
6
>
```

Here we essentially have the expression (3 + ((toEscape 3) * 4)), with our escape procedure being encapsulated within ((___ 3) * 4). When we call our escape procedure toEscape, we exit this encapsulation with the value 3, which is what we return to the level we escape to. In this instance since the procedure we are currently escape is encapsulated within (3 + _____), we will exit out this level with the number 3. This results in (3 + 3) as we escape the (___ * 4) and never encounter the computation, giving us 6.

Another powerful application of continuations is the ability to save states and essentially pause our computation and utilize it later. Similar to our toEscape example, except we set a variable to our current escape procedure, allowing us to invoke it later through our

variable. Note that is only applicable to languages with first class procedures, meaning that procedures can be passed and returned as arguments. In the example below:

```
(define setState 'Initial)

(* 2 (call/cc
      (lambda (currentState)
        (set! setState currentState) 3)))

(setState 5)
```

```
Welcome to DrRacket, version 7.2 [3m].
Language: R5RS; memory limit: 128 MB.
6
10
>
```

c with a similar example as before, except our encapsulation is with (2 * __). When we run the program, we receive 6 as a result of the initial computation as a result of (2 * 3). Here we did not escape the procedure, but rather set a variable to that state and proceeded with 3 as the value for our current level computation. However with (setState 5) we exit out of this closure, ignoring 3 and instead returning 5 to our computation. The result computation is then (* 2 5) or (2 * 5), resulting in the output 10.

This combination of escape procedures and saved states are prime examples of why continuations are so powerful in language such as Scheme, where functions and procedures are first class. In our introduction we described generators as being capable of suspending itself until it is invoked again, providing the next element each time it is called upon. This capability is done so specifically through the usage of saved states and escape procedures.

Generators Using Continuations (Pseudo)

We have gone into the concept of generators as well as the potential usages of continuations. Putting all of this together, we can begin to form a potential skeleton or idea for building a generator in Scheme. First, we note that a generator must do two things: iterate and return the next element in the structure to the procedure call. Second, suspend the generator procedure and ensure that the next call will return the *next* element in the structure. To begin developing a generator in Scheme, let us first build a list generator before moving on to a tree generator.

```
(define listInput '(1 2 3 4 5 6))
(define listGenerate (listGen listInput))
(listGenerate)
(listGenerate)
(listGenerate)
```

```
Welcome to DrRacket, version 7.2 [3m].
Language: R5RS; memory limit: 128 MB.
1
2
3
>
```

In our example code above, we utilize the list composed of numbers one through six as our input. We then invoke a generator using the input list, and then call the generator three times. The right depicts our expected input, such that we receive ‘1’, then ‘2’, then ‘3’ on our second and third call of (listGenerate). After the sixth input we would receive the empty set ‘() continuously, depending on our implementation of the generator. Ideally we would create a function that loops through it on each call, with the specifications of which computations are run on each of the elements we receive from the generator. If we are simply iterating through the list, a conditional statement to check that we have reached the end of the list should be included in the function and not the

generator. Below is a pseudocode skeleton of such a function that iterates through a list with a generator:

```
(define listLoop
  (lambda (listInput)
    (create generator using listInput)
    (call generator and define it as a variable:
          >currElement (generator)
          (cond ((null? element) '())
                (else (display currElement)
                      (loop again until null)
```

The idea is create a loop function within our listLoop, that will continuously call the list generator and assigning a variable to its value. We then perform the routine conditionally checks to ensure we have not reached the end of the list with a ‘(), and otherwise performing computations on the current element.

However, the above is for utilizing a list generator, not how to build a generator itself. To do so, we must keep in mind the important functionalities of a generator: being able to suspend its own operation and escape to a higher level with a value, and being able to continue from its suspended state when called again. This makes it clear that we need two continuations in creating a generator: one to save the state of where we are in the structure we are iterating through, as well as an escape procedure to return to with the values we find.

Once again brainstorming with pseudocode, we can see a simple skeleton of how our generator function should look:

```
(define listGenerator
  (lambda (list)
    (let ((escapeProcedure 'Initial)) ;Create an escape procedure
      (loop function to iterate througho the list:
            if null? return '()
            else:(set our current loop to this point in the procedure)
                 (escapeProcedure currentElement)
                (loop again, as we return to this position from
                      setting the state above and escaping to here)
```

It may be convenient to note that utilize the typical Scheme functions to iterate through a list is still usable here in creating our generator. Except in the generators case we do not just return the value, instead we set our function to where we are, utilize the escape procedure to escape with our value, then return to where we are by call the loop function again later. This will be done by call the generator again, which we will have invoke the loop, and then there continue from where we left off.

Now that we have an idea of how to build a list generator, it should be noted that building a tree generator is much the same in terms of structure. We utilize a standard tree walking function but instead of just returning the value and continuing to iterate, we save our state, escape with our value, then later invoke the iteration again.

Generators Implementation

Let us first look out how we implement a list generator:

```
(define listGen
  (lambda (listInput)
    (let ((escapeCont 'Initial))
      (letrec ((generate-list
                (lambda ()
                  (let loop ((list listInput))
                    (cond ((null? list) 'Skip)
                          (else (begin
                                  (call/cc
                                   (lambda(currPos)
                                     (set! generate-list currPos)
                                     (escapeCont (car list))))
                                  (loop (cdr list))))))
                  (escapeCont '()))))
        (lambda()
          (call/cc
           (lambda(toEscape)
             (set! escapeCont toEscape)
             (generate-list)))))))))
```

Let us first explain how our function works, listGen is a function which takes one argument: our list input. From there we use let in order to initialize our escape procedure in the form of **(let ((escapeCont 'Initial)).** Since we need to have an escape procedure in order to escape from our listGen function, we initialize it first we will have a nested function that will do the iterating through the list. We then utilize escapeCont to escape from the procedure and return the current element. Looking deeper into our generator function, we see that the body of our **let** is a **letrec** procedure that defines a function called **generate-list**, which is responsible for producing the elements of the list. **Generate-list** has the function **loop ((list listInput)** which is very similar to a typical walk through a list. However we see that in the conditional else that signifies an element we perform

the functionalities we mentioned earlier: setting our function generate-list to the current state in this procedure and then escaping with the **escapeCont**.

```
(let loop ((list listInput))
  (cond ((null? list) 'Skip)
        (else (begin
                (call/cc
                 (lambda(currPos)
                   (set! generate-list currPos)
                   (escapeCont (car list))))
                (loop (cdr list))))))
```

Now when we call **generate-list** again we immediately exit out of our **call/cc** and move on to **(loop (cdr list))** as that is the state we suspended at. Note that everytime we call the generator now, our **letrec** body ensures that we set the **escapeCont** to the appropriate escape procedure before calling **generate-list** again. Note that we must use **letrec** in this scenario, as we are utilizing **generate-list** within the val-expr, which is its very own definition. Since **letrec** creates the ids first, we can safely use **generate-list** within its own definition. We can now look back at our **listLoop** function and see where our elements are being sent.

```
(define listLoop
  (lambda (list)
    (let ((listGenerator (listgen list)))
      (let loop()
        (let ((leaf (listGenerator)))
          (cond ((null? leaf) '())
                (else (begin
                        (display leaf)
                        (loop)))))))))
```

```
(define listLoop
  (lambda (list)
    (let ((listGenerator (listgen list)))
      (let loop()
        (let ((leaf (listGenerator)))
          (cond ((null? leaf) '())
                (else (begin
                        (display leaf)
                        (loop)))))))))
```

We define a variable **listGenerator** as our generator in the loop function, now everytime we call **listGenerator** we receive the next element within the list and have it assigned to the variable **leaf** with **(let ((leaf (listGenerator))..)** . We then go about utilizing the current element however we wish. Next let us look at the implementation of tree generators, which shares a very similar

structure in terms of function and theory.

```
(define treeGen
  (lambda (treeInput)
    (let ((escapeCont 'Initial))
      (letrec ((generate-node
                (lambda ()
                  (let loop ((tree treeInput))
                    (cond ((null? tree) 'Skip)
                          ((pair? tree) (loop (car tree))
                                        (loop (cdr tree)))
                          (else (call/cc
                                 (lambda(currPos)
                                   (set! generate-node currPos)
                                   (escapeCont tree))))))
                  (escapeCont '()))))
        (lambda()
          (call/cc
           (lambda (toEscape)
             (set! escapeCont toEscape)
             (generate-node)))))))))
```

We see that **treeGen** shares an almost identical structure to **listGen** except for the conditionals in the loop function. This is to be expected as it still follows a generic procedure of traversing trees, calling **(loop (car tree))** and **(loop (cdr tree))**. The rest of the function follows very closely with our previously explained **listGen** function. Below is our **treeLoop** function along with some comparative function written by creating generators based on the trees given as arguments, and then individually comparing the elements returned.

```
(define treeLoop
  (lambda (tree)
    (let ((treeGener (treeGen tree)))
      (let loop()
        (let ((node (treeGener)))
          (cond ((null? node) '())
                (else (begin
                        (display node)
                        (newline)
                        (loop)))))))))
```

The below function is utilized by passing two trees in as arguments, we then generate two tree generators for each individual tree. We compare the two nodes we received and return false if they are not equal, and if they are equal, we check if we have finished iterating through the tree with null. Otherwise, we simply call loop again which then sets our node variables to the next call of our generators which yields the next nodes in our tree.

```
(define treeCompare
  (lambda (tree1 tree2)
    (let ((treeGen1 (treeGen tree1))
          (treeGen2 (treeGen tree2)))
      (let loop()
        (let ((node1 (treeGen1))
              (node2 (treeGen2)))
          (cond ((eq? node1 node2)
                 (cond ((null? node1) #t)
                       (else (loop))))
                (else #f)))))))
```

The below **findElem** function takes a tree and an element (integer) as input, this will create a generator based on our tree argument and compare each node we obtain from our generator to the **elem** argument we passed.

```
(define findElem
  (lambda (tree1 elem)
    (let ((treeGener (treeGen tree1)))
      (let loop()
        (let ((node (treeGener)))
          (cond ((null? node) #f)
                (else (cond ((eq? node elem) #t)
                            (else (loop))))))))))
```

We used **define-syntax** in order to create a for loop variation written in Scheme:

```
(define-syntax for
  (syntax-rules (in)
    ((_ node in (tree) body ...)
     (let ((treeGener (treeGen tree)))
       (let loop()
         (let ((node (treeGener)))
           (cond ((null? node) '())
                 (else (begin
                         body ...
                         (loop)))))))))))
```

```
(for node in (treel)
  (display node)
  (newline))
```

Once we understand how **define-syntax** operates, this becomes rather trivial with our tree generator. The for after **define-syntax** determines the start of the pattern we are looking for when defining our for loop. The ‘in’ in **syntax-rules(in)** is another literal-id we take into account when determining our pattern. We utilize the form (**head args) body … +** for our define syntax, taking in **(tree)** which is our tree input and using it to create a generator. We then set **node** to the node we obtain from the generator, and pass it through **body …** which in this case is a simple display and newline function, before calling loop again to continue iterating.

Conclusion

While generators have been mostly replaced by the contemporary for loops and iterators we typically find nowadays in modern languages, they are still powerful tools that can be more efficiently depending on the type of functionality required. Continuations with their ability to manipulate the control stack and allowing programmers to more effectively direct their procedures and functions also have great usages. In fact, exceptions are an example of continuations and are almost universally present in all modern languages.

We also saw how to create a generator in Scheme, but the concept and approach is applicable to all languages that support continuations ands first class procedures/functions. Much of the iterating is done similar to how one would typically walk through a structure. However the core components of a generator is the ability to suspend, resume, and save its operation status, along with escaping with a desired value once found. Which is all made aptly convenient to implement using continuations for escape procedures and save states.

References

1. Continuations Implement Generators and Streams

   By: L. Allison, 01 January 1990

   https://doi.org/10.1093/comjnl/33.5.460

2. Continuations, Functions, and Jumps

   By: Hayo Thielecke, June 1999

   https://dl.acm.org/citation.cfm?id=568561

3. The Discoveries of Continuations

   By: John C. Reynolds, 1993

   https://homepages.inf.ed.ac.uk/wadler/papers/papers-we-love/reynolds-discoveries.pdf

4. Applications of Continuations

   By: Daniel P. Friedman, 1988

   https://cs.indiana.edu/~dfried/appcont.pdf

Appendix

```
(define call/cc call-with-current-continuation)

;List Generator

(define listGen
  (lambda (listInput)
    (let ((escapeCont 'Initial))
      (letrec ((generate-list
                (lambda ()
                  (let loop ((list listInput))
                    (cond ((null? list) 'Skip)
                          (else (begin
                                  (call/cc
                                   (lambda(currPos)
                                     (set! generate-list currPos)
                                     (escapeCont (car list))))
                                  (loop (cdr list))))))
                  (escapeCont '()))))
        (lambda()
          (call/cc
           (lambda(toEscape)
             (set! escapeCont toEscape)
             (generate-list)))))))))

(define listLoop
  (lambda (list)
    (let ((listGenerator (listgen list)))
      (let loop()
        (let ((leaf (listGenerator)))
          (cond ((null? leaf) '())
                (else (begin
                        (display leaf)
                        (loop)))))))))

;Below are test cases for our list generator

(define list1 '(1 2 3 4 5 6 7))
```

```
(display 'Start_of_listLoop_Example)
(newline)

(listLoop list1)

(display 'End_of_listLoop_Example)
(newline)


;Tree generator (iterates through nodes)

(define treeGen
  (lambda (treeInput)
    (let ((escapeCont 'Initial))
      (letrec ((generate-node
                (lambda ()
                  (let loop ((tree treeInput))
                    (cond ((null? tree) 'Skip)
                          ((pair? tree) (loop (car tree))
                                        (loop (cdr tree)))
                          (else (call/cc
                                 (lambda(currPos)
                                   (set! generate-node currPos)
                                   (escapeCont tree))))))
                  (escapeCont '()))))
        (lambda()
          (call/cc
           (lambda (toEscape)
             (set! escapeCont toEscape)
             (generate-node)))))))))

(define treeLoop
  (lambda (tree)
    (let ((treeGener (treeGen tree)))
      (let loop()
        (let ((node (treeGener)))
          (cond ((null? node) '())
                (else (begin
                        (display node)
                        (newline)
                        (loop)))))))))
```

```
;Test cases for tree generator

(define tree1 '(((2)3)5((6(7))8(9))))

(define tree2 '(((2)3)5((6(11))8(9))))

(display 'Start_of_treeLoop_example)
(newline)

(treeLoop tree1)

(display 'End_of_treeLoop_example)
(newline)


;Using two generators to compare two trees

(define treeCompare
  (lambda (tree1 tree2)
    (let ((treeGen1 (treeGen tree1))
          (treeGen2 (treeGen tree2)))
      (let loop()
        (let ((node1 (treeGen1))
              (node2 (treeGen2)))
          (cond ((eq? node1 node2)
                 (cond ((null? node1) #t)
                       (else (loop))))
                (else #f)))))))

;(treeCompare tree1 tree1)
;(treeCompare tree1 tree2)

;Finding an element in tree using our generator
(define findElem
  (lambda (tree1 elem)
    (let ((treeGener (treeGen tree1)))
      (let loop()
        (let ((node (treeGener)))
          (cond ((null? node) #f)
                (else (cond ((eq? node elem) #t)
                            (else (loop)))))))))))
```

```
;(findElem tree1 6)
;(findElem tree1 1)


;Using our generator in order to create a for loop syntax
(define-syntax for
  (syntax-rules (in)
    ((_ node in (tree) body ...)
     (let ((treeGener (treeGen tree)))
       (let loop()
         (let ((node (treeGener)))
           (cond ((null? node) '())
                 (else (begin
                         body ...
                         (loop)))))))))

;Testcase for our for loop

(for node in (tree1)
  (display node)
  (newline))
```