Call with Current Continuation Patterns

Darrell Ferguson [*] Dwight Deugo[⋆]

August 24, 2001

Abstract

This paper outlines the recurring use of continuations. A brief overview of continuations is given. This is followed by several patterns that outline the use of continuations leading up to using continuations to implement coroutines, explicit backtracking, and multitasking. Scheme is used for the examples as it supports first class continuations.

1 Introduction

We often find ourselves in a situation while researching a selected topic (say the use of continuations in Scheme) where we have found a paper that deals with the topic and read a portion of the paper only to realize that our knowledge is not extensive enough to understand all the material. So, we stop reading the paper and begin reading the papers that are used as references and other material that gives us some background knowledge of the subject. After reading the background work, we return to read the rest of the original paper. Well, just as we packaged up our reading of the first paper and returned to it at a later time, continuations allow for the same ability in programming.

Although they do not fit into what most people see as standard programming (procedural) and they are foreign to most programmers, continuations allow for a variety of complex behaviours without modification to the interpreter or compiler. Powerful control structures can be implemented using only continuations including multitasking, backtracking, and several escape mechanisms.

Unfortunately the concept of first class continuations (continuations that can be passed as arguments, returned by procedures, and stored in a data structure to be returned to later) does not exist in most languages. This is why Scheme [Dyb96] was used as the language for our examples of the usage of continuations.

We will begin by providing a brief introduction to continuations for those who are not familiar with them. In the next section we will describe two concepts - *contexts* and *escape procedures*- that will allow us to understand continuations. We will then show how continuations can be made from these two concepts. We will then give several patterns which

[*]School of Computer Science, Carleton University. darrell.ferguson@ottawa.com deugo@scs.carleton.ca . Copyright © 2001, Darrell Ferguson and Dwight Deugo. Permission is granted to make copies for purposes related to the 2001 Pattern Languages of Programs Conference. All other rights reserved.

demonstrate the recurring use of continuations ending with their use in coroutines[HFW86], explicit backtracking [FHK84] [SJ75] and multitasking [DH89].

1.1 Continuations

We will begin our discussion of continuations by defining the context of an expression as well as escape procedures. Our examples are taken from [SF89] and more information about these two concepts can be seen there.

1.1.1 Contexts

In [SF89] the *context* is defined as a procedure of one variable, □. To obtain the context of an expression we follow two steps; 1) replace the expression with □ and; 2) we form a procedure by wrapping the resulting expression in a lambda of the form (lambda (□) ...). For example, we can take the context of (+ 5 6) in (+ 3 (* 4 (+ 5 6))) to be:

```
(lambda (□)
      (+3 (* 4 □)))
```

We can extend this by extending the first step of context creation. We can evaluate the expression with □ and when evaluation can no longer proceed (because of □), we will have finished the first step. To demonstrate this we will look at finding the context of (* 3 4) in:

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ (* 3 4) 5) 2))
```

We begin by replacing (* 3 4), giving us:

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ □ 5) 2))
```

Continuing with the first step, we evaluate (zero? 5) to be false and choose to calculate the alternative part of the if statement. This gives us (* (+ □ 5) 2). No more computation can take place so we continue to the second step and the context becomes:

```
(lambda (□)
      (* (+ □ 5) 2))
```

1.1.2 Escape Procedures

The *escape procedure* is a new type of procedure. When an escape procedure is invoked, its result is the result of the entire computation and anything awaiting the result is ignored.

The **error** procedure is an example of an escape procedure where everything that is waiting for the computation is discarded and an error message is immediately reported to the user.

Now let us assume that there exists a procedure **escaper** which takes any procedure as an argument and returns a similarly defined escape procedure [SF89]. An example of the **escaper** procedure would be:

```
(+ ((escaper *) 5 2) 3)
```

The expression (escaper *) returns an escape procedure which accepts a variable number of arguments and multiplies these arguments together. So, when ((escaper *) 5 2) is invoked, the waiting + is abandoned (due to the escape procedure) and 10 is returned [the result of (* 5 2)].

In *Escape from and Reentry into Recursion* we will see a definition of the **escaper** function that is made using continuations.

1.1.3 Defining Continuations

The procedure **call-with-current-continuation** (which is usually shortened to **call/cc**) is a procedure of one argument (we will refer to this argument as a *receiver*). The *receiver* must be a procedure of one argument which is called a *continuation.*

The **call/cc** procedure forms a continuation by first determining the context of **(call/cc receiver)** in the expression. The escaper procedure shown in the previous section is then invoked with the context as an argument and this forms the continuation. The receiver procedure is then invoked with this continuation as its argument.

Example

If we take for example the expression:

```
(+ 3 (* 4 (call/cc r)))
```

The context of (call/cc r) is the procedure:

```
(lambda (□) (+ 3 (* 4 □)))
```

So the original expression can be expanded to:

```
(+ 3 (* 4 (r (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

If we consider **r** to be **(lambda (continuation) 6)** the above works out to be:

```
(+ 3 (* 4 ((lambda (continuation) 6)
           (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

The continuation (and escape procedure) is never used so:

```
((lambda (continuation) 6) (escaper (lambda (□) (+ 3 (* 4 □)))))
```

results in 6 and the entire expression returns 27 (3 + 4 * 6). However if **r** was **(lambda (continuation) (continuation 6))**, we would have:

```
(+ 3 (* 4 ((lambda (continuation) (continuation 6))
           (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

Since we invoke continuation on 6, we have:

```
((escaper (lambda (□) (+ 3 (* 4 □)))) 6)
```

Remember that the escaper procedure returns an escape procedure which abandons its context (so the + and * waiting for the result of evaluating the escaper function are discarded). Although this still results in 27, the important point is the process of getting the result has been changed.

2 The Patterns

Escape from a Loop

This pattern demonstrates the separation of code into a control structure (looping mechanism) and action (what to perform during each loop). Continuations are used for breaking out of the control structure. We use call/cc and a receiver function to get the escape procedure before entering the loop. Now the body of the loop has access to a function which will escape its context, breaking the loop.

Context

We have a system that can be looked at as a combination of two constructs, the control (looping mechanism) and action (what to perform during each loop).

Problem

How do we break from our looping mechanism?

Forces

- Delegation of looping to a single reusable function allows for code reuse but stopping could be complicated.
- Copying the looping mechanism and merging it with each action results in duplicating code.
- Attempting to check the return value inside the loop mechanism each time the action is performed looking for a special return value to signal stopping the loop restricts what the function can return. If we change this special return value we must change every function that is passed into it.
- We want to keep knowledge of how the looping mechanism works away from the action to ensure that these parts are independent of each other.

Solution

We can use a continuation to store the context before entering the loop and use it as an escape procedure when the exit condition is met. We begin our procedure by creating a receiver procedure for a continuation (which we will use as our escape procedure). This receiver procedure will invoke the procedure that contains the looping mechanism with the action procedure to be invoked at each step passed in as an argument. This action procedure will be defined within the receiver procedure to enable access to the exit-procedure (due to the lexical scoping). Terminating the infinite loop can be accomplished by invoking the escape procedure (the continuation) inside the single action procedure with whatever exiting value is wanted when the exiting condition(s) is/are met.

Sample Code

An infinite loop can be made by creating a procedure that takes a procedure as an argument. This function would have a locally defined looping function that 1) calls the original procedure passed in; and 2) recursively calls itself. We could also place any code that should be executed during each loop (before or after the action) in this procedure. With these parts shown in italics, the looping procedure would be:

```
(define infinite-loop
    (lambda (procedure)
        (letrec ((loop (lambda ()
                       ... code to execute before each action ...
                       (procedure)
                       ... code to execute after each action ...
                       (loop))))
            (loop))))
```

We will now look at the structure of an action procedure with the parts of the procedure to be filled in shown in italics.

```
(define action-procedure
    (lambda (args )
        (let ((receiver (lambda (exit-procedure)
                            (infinite-loop
                                (lambda ()
                                    (if exit-condition-met
                                        (exit-procedure exit-value )
                                        action-to-be-performed )))))))
              (call/cc receiver))))
```

Example

Using the definition of **infinite-loop** with no code added before or after the action, we can create a procedure which counts to **n** (displaying each number it counts). At **n**, the function will escape the loop and return *n*.

```
(define count-to-n
    (lambda (n)
      (let ((receiver (lambda (exit-procedure)
                          (let ((count 0))
                              (infinite-loop
                                  (lambda ()
                                      (if (= count n)
                                          (exit-procedure count)
                                          (begin
                                              (write-line ”The count is: ”)
                                              (write-line count)
                                              (set! count (+ count 1))))))))))
            (call/cc receiver))))
```

Rationale

We have now broken up our procedure into two parts, the control (loop) and the action. Using call/cc, we have a way of stopping the infinite loop. Since our looping mechanism has been separated from our action we are able to use the same looping mechanism with different action procedures and we can add default behaviour to be performed during each iteration of the loop without having to modify each action procedure.

Escape from Recursion

This pattern presents a solution to escaping from a recursive computation. Similar to the *Escape from a Loop* , we create an escape procedure in the context before entering the recursive computation and use this procedure if a break condition is met.

Context

We are writing a recursive procedure. Our domain makes it possible that we will know the final result without completing the computation.

Problem

How do we escape from a recursive computation?

Forces

- Exiting out of a computation (discarding the computation that has been built up) will make the procedure extremely efficient for special cases (possibly avoiding all computation).
- You may lose readability by adding complexity to the procedure.
- Building a recursive computation will also temporarily build up stack space which can normally be avoided with a language that supports tail-recursion.

Solution

We can use **call/cc** and a receiver function to create an escape function with the context before any of the computation is recursively built up. We can then perform the normal recursive function using a recursive helper function defined within the receiver function (to provide access to the continuation). If the special condition is met, the continuation is invoked with the desired exit value.

Sample Code

We will now look at the structure of a recursive procedure using **call/cc** to exit out of the recursive computation. The portions of the procedure to be filled in are shown in italics.

```
(define function
    (lambda (args )
        (let ((receiver
                (lambda (exit-procedure)
                    (letrec ((helper-function
                                (lambda (args )
                                    (cond
                                        (break-condition (exit-function exit-value ))
                                        other cases and recursive calls
                          (helper-function args )))))
                    (call/cc receiver))))
```

Examples

If we need to compute the product of a non-empty list of numbers, we know that if 0 occurs in the list the product will be 0. So, we can write product-list as:

```
(define product-list
    (lambda (nums)
        (let ((receiver
                (lambda (exit-on-zero)
                    (letrec ((product
                                (lambda (nums)
                                    (cond
                                        ((null? nums) 1)
                                        ((zero? (car nums)) (exit-on-zero 0))
                                        (else (* (car nums)
                                                (product (cdr nums))))))))
                            (product nums)))))
        (call/cc receiver))))
```

If we take (product-list '(1 2 3 0 4 5)) we get:

$\Longrightarrow$ (product '(1 2 0 3 4))
$\Longrightarrow$ (* 1 (product '(2 0 3 4)))
$\Longrightarrow$ (* 1 (* 2 (product '(0 3 4))))
And at this point, the exiting condition is met and we exit from the computation and return 0. Notice that the waiting multiplications were discarded.

We can apply this to deep recursion (tree recursion) as well. If we allow the list of numbers to contain lists of numbers and so on, we can rewrite product-list to be:

```
(define product-list
    (lambda (nums)
        (let ((receiver
                (lambda (exit-on-zero)
                    (letrec ((product
                                (lambda (nums)
                                    (cond
                                        ((null? nums) 1)
                                        ((number? (car nums))
                                            (if (zero? (car nums))
                                                (exit-on-zero 0)
                                                (* (car nums)
                                                    (product (cdr nums)))))
                                        (else (* (product (car nums))
                                                (product (cdr nums))))))))
                        (product nums)))))
                (call/cc receiver))))
```

Rationale

Using **call/cc** to escape from the recursive process adds little code to the original recursive procedure. It also allows us to discard much of the unnecessary computation (sometimes all of the computation can be avoided).

Consequences

A language that supports tail-recursion will allow you to write recursive functions that do not build up the stack (tail-recursive procedures). A tail-recursive procedure will keep a running result of the computation. This means that this pattern has little effect when escaping from a tail-recursive procedure. However, if the computation at each recursive call is expensive and the procedure will not build up a huge computation stack, this pattern can be used as an alternative to tail-recursion.

Loop via Continuations

This pattern presents a solution to creating a loop using continuations. We use call/cc to get an escape procedure which will return to the context of the beginning of the loop body. We can then use this escape procedure to escape from the current context back to the beginning of the body which we want to loop over.

Context

Although the *Escape from a Loop* pattern deals with escaping from an infinite looping mechanism, we can also face the situation where we want the ability to loop over a portion of a procedure or computation (which may be separated over portions of several procedures) until some condition is met.

Problem

How do we loop until condition is met using continuations?

Forces

- Other existing looping mechanisms such as the **for** or **while** loops are more common and are what programmers are used to.
- The existing **for** and **while** loops have to be contained within one procedure.
- Separating the mechanism for looping over several procedures may make the program more difficult to read.
- We may be able to write our program quicker or more easily by writing the code to perform one iteration without worrying about looping and adding the loop mechanism later.
- Looping can be added to portions of existing code with minimal effort and changes to the existing program.

Solution

We can create a loop by getting the continuation at the beginning of the portion of the procedure to be looped over and storing it in a temporary variable. If we determine that looping becomes necessary, we simply invoke the continuation. This will escape the current context and return to the beginning of the loop portion. Getting a handle on a continuation can be done easily with the identity function using the line **(call/cc (lambda (proc) proc))**.

For terminating the loop we will need some way of changing the state of the execution from a state which will continue the loop to one that will not (ie. the loop-condition becomes false). This can be done by the manipulation of variables which are defined outside of the loop. We can then use assignment (**set!**) to change this value. Another approach would be to add the current values to the continuation's argument.

When the portion of code to be looped over is separated amongst several procedures where we want to exit out of one of these procedures and start the loop over, we can simply pass the continuation along as an argument. Inside these functions, if the condition to return to the beginning of the loop is met, we simply invoke the continuation with itself as a parameter.

Sample Code

We will now look at the structure of the loop using **call/cc** . The portions of the procedure to be filled in are shown in italics.

```
(define partial-loop
    (lambda (args )
          ... preliminary portion ...
          (let ((continue (call/cc (lambda (proc) proc))))
                 ... loop portion ...
                 (if loop-condition
                     (continue continue)
                     ... final portion ... )))
```

The reason for invoking **(continue continue)** to loop is tricky. If we remember how **call/cc** works, we first get the context of the **call/cc** call. Here, that would be:

```
(lambda (□)
     (let ((continue □))
            ... loop portion ...
            (if loop-condition
                (continue continue)
                ... final portion ... )))
```

Examining this, when we invoke **(continue **some argument**)** , whatever argument we apply will now become the value of **continue** . Since we are attempting to loop by continually returning to this point of execution it seems logical that we do not wish for this value to change so we pass the previous value of **continue** (which is our continuation) to be rebound to **continue** allowing us to repeatedly return to the same point of execution.

Example

Here is an example where we are given a non-empty list of numbers and we want to increment each element of the list by 1 until the first element is greater than or equal to 100. We will separate out the initial construction of the continuation to a separate function (labelled **get-continuation-with-values**) for readability.

```
(define get-continuation-with-values
    (lambda (values)
         (let ((receiver (lambda (proc) (cons proc values))))
             (call/cc receiver))))

(define loop-with-current-value
    (lambda (values)
        (let ((here-with-values (get-continuation-with-values values)))
            (let ((continue (car here-with-values))
                (current-values (cdr here-with-values)))
                    (write-line current-values)
                    (if (< (car current-values 100))
                        (continue (cons continue
                                        (map (lambda (x) (+ x 1))
                                            current-values)))
                        (write-line "Done!"))))))
```

If we attempted **(loop-with-current-value '(1 2 3))**, the first value of **here-with-values** would be the list (*continuation* 1 2 3). We then separate this into **continue** being *continuation* and **current-values** being '(1 2 3). The first time we execute the line **(continue (cons continue (map (lambda (x) (+ x 1)) current-values)))**, we would be rebinding **here-with-values** to be (*continuation* 2 3 4). We then hit that line again and go back to rebind **here-with-values** to (*continuation* 3 4 5). This continues until we get the **here-with-values** bound to (*continue* 100 101 102) at which point we write the string "Done!" to the screen and exit.

Rationale

The use of continuations to create the looping mechanism adds little code to create the loop and with a few comments in the code saying how the continuations are being used (to return to the beginning of the loop body) the program remains readable. With the code broken up into procedures, code can be reused in different loops that use this looping mechanism (passing in their own continuation to these procedures).

Escape from and Reentry into Recursion

This pattern presents a solution to the problem of escaping from a recursive process while allowing for the process to be reentered. We create a break procedure which will first store the current continuation in a specified scope and then escape from the recursive process. Invoking the stored continuation will continue the recursive process.

Context

We have a recursive computation (possibly deep or tree recursion) which we want to escape from but keep the ability to go back and continue the execution. This will allow for a programmer to debug his/her code by determining when special conditions are met. We can also use this tool to change the internal values of the function or possibly some external variables as the computation is progressing so that we can change the computation on the fly. For this pattern, the reentering of the computation can be instigated by the user.

Problem

How do you escape from a recursive computation while allowing for the ability to reenter at the point of exit?

Forces

- We want to minimize the amount of code modified and added while keeping the code readable. This will allow for this feature to be removed easily if it is being done for debugging purposes.
- We want to avoid modifications to the interpreter/compiler in adding this ability (we do not want a full debugging facility).

Solution

We will use for our solution, the **escaper** procedure from [SF89] to escape the current computation and **call/cc** to store the context that we are breaking from so that it can be reentered. We will need a **break** procedure as well as a **resume-computation** procedure, both of which will need to be globally accessible. The **break** procedure should do two things; 1) store the current continuation and 2) stop the current computation. The **break** procedure can also take in a value which will be passed in to the continuation when it is resumed. The **resume-computation** procedure will, as the name says, resume the currently escaped computation. If we have these functions it should be a simple matter of insterting a break when we wish to halt execution.

In creating the **break** procedure we will begin by looking at the second requirement. We have already discussed escape procedures and [SF89] gives a definition of the **escaper** procedure. We can use this procedure to escape the current computation. As for first part of the break, we need to begin by getting the current continuation (done simply through the use of call/cc with a receiver function). Looking ahead at the **resume-computation** procedure, we can see that invoking this procedure should invoke the continuation that **break** stored with the argument that was passed in originally. So, we can have **break** set the **resume-computation** procedure to be **(lambda () (continuation arg))**.

Sample Code

The break procedure would be:

```
(define break
    (lambda (arg)
        (let ((exit-procedure
                    (lambda (continuation)
                        (set! resume-computation (lambda () (continuation arg)))
                        (write-line ”Execution paused. Try (resume-computation)”)
                        ((escaper (lambda () arg))))))
            (call/cc exit-procedure))))
```

However, we will need to initialize **resume-computation** to some temporary procedure (which will be overwritten the first time that break is invoked). So we will define it to be **(lambda () ”to be initialized”)**.

We will now create the **escaper** procedure mentioned in Section 1.1.2. To begin, we will need to store a continuation (which we will call **escape-thunk**) with the context **(lambda (□) (□))**. This can be done by first assigning it to some procedure:

```
(define escape-thunk (lambda () ”escape-thunk Initialized”))
```

We then need to create a receiver function for a continuation which will assign the continuation to be our **escape-thunk**.

```
(define escape-thunk-init
    (lambda (continue)
        (set! escape-thunk continue)))
```

And we simply need to create the continuation using **call/cc** with this receiver function.

```
((call/cc escape-thunk-init))
```

The **escaper** procedure can then be defined as:

```
(define escaper
    (lambda (procedure)
        (lambda args
            (escape-thunk (lambda () (apply procedure args))))))
```

Example

An example of this pattern would be the flatten-list procedure below which will escape every time a non-list argument is passed to the function.

```
(define flatten-list
    (lambda (arg)
        (cond
            ((null? arg) '())
            ((not (list? arg)) (list (break arg)))
            (else
                (append (flatten-list (car arg))
                    (flatten-list (cdr arg)))))))
```

And the output of running **(flatten-list '(1 2 3))** will be:

```
1 ]=> (flatten-list '(1 2 3))
"Execution paused. Try (resume-computation)"
;Value: 3
1 ]=> (resume-computation)
"Execution paused. Try (resume-computation)"
;Value: 2
1 ]=> (resume-computation)
"Execution paused. Try (resume-computation)"
;Value: 1
1 ]=> (resume-computation)
;Value 3: (1 2 3)
```

Coroutines

Coroutines allow for sequential control between procedures. Using call/cc, we obtain the current context of a procedure and store it so it can later be invoked.

Context

The process of many programs can be viewed as the passing of control sequentially among several entities. A good analogy is to compare this to many traditional card games where each player takes his/her turn and then passes control to the next person.

Problem

How do we allow sequential control among several entities?

Forces

- Having each entity's process in one procedure allows us to view each process in one place.
- Breaking each entity's process up into many small procedures that force the sequential control becomes difficult to trace through for the programmer and difficult to change the process of control.
- Using a mechanism such as threads and semaphores may have a high overhead or may not be available.
- The switching of threads is traditionally done at a lower level and to design the threads to wait until called upon to act would be complex and hard to follow.

Solution

We can use continuations to implement coroutines. Coroutines allow for interruption of a procedure as well as resuming an interrupted procedure. We will begin by creating a coroutine for each entity that will share control. The **coroutine-maker** (as defined in [SF89]) will take as its argument a procedure that represents the body or actions of the entity. This *body-procedure* will take two arguments, a *resumer* procedure that will be used to resume the next coroutine in order and an initial value.

The **coroutine-maker** will create a private **update-continuation** function to be used to store the current continuation of the procedure. Each time the coroutine is invoked with a value, it passes that value to its continuation. When a coroutine passes control to another coroutine, it updates its current state to its current continuation and then resumes the second coroutine.

Sample Code

The **coroutine-maker** from [SF89] is:

```
(define coroutine-maker
    (lambda (proc)
        (let ((saved-continuation ’()))
            (let ((update-continuation! (lambda (v)
                                            (write-line ”updating”)
                                            (set! saved-continuation v))))
                (let ((resumer (resume-maker update-continuation!))
                    (first-time #t))
                    (lambda (value)
                        (if first-time
                            (begin
                                (set! first-time #f)
                                (proc resumer value))
                            (saved-continuation value))))))))
```

As you can see, **coroutine-maker** makes use of a helper function **resume-maker**. Again from [SF89], **resume-maker** will be defined as:

```
(define resume-maker
    (lambda (update-proc!)
        (lambda (next-coroutine value)
            (let ((receiver (lambda (continuation)
                                (update-proc! continuation)
                                (next-coroutine value))))
            (call-with-current-continuation receiver)))))
```

Example

A simple example of coroutines that make use of continuations can be found in [SF89]. A more advanced mechanism and several extensions to coroutines can be found in [HFW86].

Using the **coroutine-maker** from [SF89] we can create two procedures labelled **ping** and **pong** which will switch control back and forth three times. The **ping-procedure** and **pong-procedure** will be the *body-procedures* (used by the **coroutine-maker** to create the coroutines) for **ping** and **pong** respectively.

```
(define ping
    (let ((ping-procedure (lambda (resume value)
                                (write-line ”Pinging 1”)
                                (resume pong value)
                                (write-line ”Pinging 2”)
                                (resume pong value)
                                (write-line ”Pinging 3”)
                                (resume pong value))))
```

```
                (coroutine-maker ping-procedure)))

   (define pong
       (let ((pong-procedure (lambda (resume value)
                                  (write-line ”Ponging 1”)
                                  (resume ping value)
                                  (write-line ”Ponging 2”)
                                  (resume ping value)
                                  (write-line ”Ponging 3”)
                                  (resume ping value))))
                (coroutine-maker pong-procedure)))
```

And the output of running **(ping 1)** will be:

```
1 ]=> (ping 1)

”Pinging 1”
”Ponging 1”
”Pinging 2”
”Ponging 2”
”Pinging 3”
”Ponging 3”
;Value: 1
```

Rationale

The use of continuations to create coroutines allow us to write the process for each entity in one procedure which makes the program more readable and easier to modify. The use of continuations enable us to achieve the same result as threads without modification to the programming language or interpreter.

Non-blind Backtracking

This pattern addresses the issue of allowing us to move to a previous or ”future” point of execution. Continuations are stored for past and future points and ”angel” and ”devil” procedures are given which allow us to move to a future point of execution or a past point respectively.

Context

We have reached a point in the computation where we wish to stop or suspend the current process and jump to a previous point in the computation.

Problem

How do we construct a mechanism which will allow us to get back to a past point in execution and follow another path to determine the solution?

Forces

- We need a simple mechanism for moving back and forth in a computation that is robust enough to handle situations where it may not be enough to simply go back and forth.
- Modifying or redesigning a program to incorporate the mechanism should be simple and not add too much complexity to the existing code that would make it unreadable.
- Designing the system to achieve a form of backtracking through breaking the process into recursive process that achieve backtracking by returning back to the calling procedure would be inefficient if we want to return to a point that was reached very early in the computation. This would also be very difficult to follow for the programmer for complex backtracking programs.

Solution

We can implement non-blind backtracking using **call/cc** to get escape procedures to various contexts. We store these continuations in a global structure so that they can be invoked and the process will return to these contexts in the computation.

In [FHK84] the concept of *devils*, *angels* and *milestones* is presented. A devil will return us to the context in which the last milestone was created. The devil will pass to the continuation the value that was passed to it. Now this value is used as if it were the result of the original milestone expression, possibly allowing us to follow a different path. An angel will send the computation forward to the last encounter with a devil. Again, the value passed to the angel will be given to the devil’s continuation allowing us to return to the context of the devil with this value replacing the value returned by the devil. This will allow us to move to more advanced states. A milestone will record the current context to be used by any encountered devils.

As a metaphor, we can take the example given in the introduction of this pattern language. We begin by reading the first paper and we reach a point where we realize that we need further knowledge. We set a milestone (remembering where we were in this first paper) and begin reading the references and other material. When we feel that we have sufficient knowledge to continue with the original paper, we return to that paper (equivalent to invoking a devil). Possibly, we didn't read all the references and other related material before we went back to this original paper. If this is the case, after finishing the original paper we decide to go back to reading the remaining references and other material. This is equivalent to invoking an angel.

We also need two datastructures for storing past milestones and future points (where devils were invoked). A simple approach would be to keep two stacks (past and future) since in the simplest case we will only be returning to the previous milestone or the previous invocation of a devil. Other data structures are possible and are often used in artificial intelligence applications [FHK84].

Sample Code

An implementation of milestone, angels and devils can be seen in [FHK84]. The milestone procedure should simply store the current state in the past entries and return the initial value.

```
(define milestone
    (lambda (x)
        (call/cc (lambda (k)
                    (begin (push past k)
                        x)))))
```

The implementation of a devil will store the current continuation as a future and return to the last milestone with a new value.

```
(define devil
    (lambda (x)
        (call/cc (lambda (k)
                    (begin (push future k)
                        ((pop past) x))))))
```

And the angel procedure can be written as:

```
(define angel
    (lambda (x)
        ((pop future) x)))
```

And for each of these procedures if the corresponding stack is empty it is set up to return the identity function (and the angel or devil procedure returns x).

Rationale

We can now use these procedures to store important points in the execution to be returned to, return to stored states to continue computation and return to computations that were only partially complete. With these procedures, we have a simple isolated mechanism which can be reused in various programs. We can also expand the behaviour by changing the data structure used by the backtracking mechanism from a stack to another structure.

Multitasking

Multitasking allows us to enforce the amount of time that a computation can run. Using engines and a timer mechanism we allow for processes to be interrupted (and possibly restarted). Engines use call/cc to get the current context of a computation which is invoking the engine (so that we may return to it after execution is complete) and to get the current continuation of a procedure when it is being interrupted so that it can be continued.

Context

It is often beneficial to perform distinct computations in a parallel fashion even if they are not exactly in parallel. There are also times when we wish to limit the amount of time a computation can take. An example of these would be the *or* of several expressions. Computing the expressions in parallel, if a simple expression quickly evaluates to be true, the computation of the other expressions can be terminated.

Problem

How do we perform this multitasking? This would include allowing for computations to run for a limited period of time, interrupted if they do not complete, and later restarted.

Forces

- Modifications to the interpreter/compiler or language should be kept to a minimum.
- Using a mechanism such as threads and semaphores may have a high overhead or may not be available.
- Performing the scheduling and state mechanisms in the language will allow for flexibility but is less efficient than a lower level mechanism.

Solution

We can wrap a procedure in an *engine* (one can think of an engine as similar to a thread). The engine implementation will use call/cc to: 1) obtain the continuation of the computation that invokes the engine so it can be returned to when the engine is complete, and 2) obtain the continuation of the engine when the timer expires so that it is possible to return to the computation if called upon [DH89].

The **make-engine** procedure must take a procedure (of no arguments) which specifies the computation to be performed by the engine. The engine returned will be a procedure of three arguments: 1) the amount of time the engine will be allowed to run, 2) a return procedure which specifies what to do if the engine is done before the time allotted, and 3) an expire procedure which specifies what to do if the time allotted has run out before the computation.

We will not discuss the timing mechanism as a method is given in [DH89] for creating a timer mechanism by overwriting the **lambda** function or by creating a *timed-lambda* function and using it to define all asynchronous functions. We will assume that this timer given in [DH89] is used. With this timer, a tick is consumed by an engine on every function call.

Example

We can make use of **extend-syntax** (Scheme's macro facility), to create a **parallel-or** which uses engines as (taken from [DH89]):

```
(extend-syntax (parallel-or)
          ((parallel-or e ...)
          (first-true (lambda () e) ...)))

(define first-true
        (lambda proc-list
              (letrec ((engines (queue))
                    (run (lambda ()
                               (and (not (empty-queue? engines))
                                    ((dequeue engine)
                                       1
                                       (lambda (v t) (or v (run)))
                                       (lambda (e) (enqueue e engines) (run)))))))
                    (for-each (lambda (proc) (enqueue (make-simple-engine proc) engines))
                          proc-list)
                    (run))))
```

Rationale

With continuations, no modifications are needed to the interpreter or compiler (although we do need to make use of the macro facility). There is also a small overhead of timer updates but we can avoid some of these by making some of the code synchronous. We do not need thread nor semaphores built into the language.

3 Summary

We have presented in this pattern language the idea of first class continuations and several repetitive uses of continuations from non-local exits (with the *Escape from a Loop* and *Escape from Recursion* patterns), to control structures (the *Looping with Continuations* and *Escape from and Reentry into Recursion* patterns) to the more complex behaviours (*Coroutines*, *Backtracking*, and *Multitasking* patterns). Although continuations are often confusing and hard to understand (and should normally be avoided for simpler programs), they present us with the ability to add complex behaviours that would otherwise require modification to the language, interpretor and/or compiler (provided we have the means to add to the language).

4 Bibliography

References

[DH89] R. K. Dybvig and R. Hieb. Engines from Continuations. *Computer Languages*, pages 109–123, 1989.

[Dyb96] R. K. Dybvig. *The Scheme Programming Language*. Prentice Hall, 1996.

[FHK84] D. P. Friedman, D. T. Haynes, and E. E. Kohlbecker. Programming with Continuations. *In P. Pepper, editor, Program Transformation and Programming Environments*, pages 263–274, 1984.

[HFW86] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining Coroutines with Continuations. *Computer Languages*, pages 143–153, 1986.

[SF89] G. Springer and D. P. Friedman. *Scheme and the Art of Programming*. MIT Press and McGraw-Hill, 1989.

[SJ75] G. J. Sussman and G. L. Steele Jr. Scheme: An Interpreter for Extended Lambda Calculus. MIT AI Memo 349, Massachusetts Institute of Technology, May 1975.