Variation on a Scheme: multiple implicit first-class continuations

and a general attempt at understanding continuations

Jonathan Lam

August 23, 2021

1 Introduction

This is for the completion of ECE491: Structure and Implementation of Computer Programs, based on the text of the same name by Abelson and Sussman. The assignment was to implement an extension of the interpreter presented in sections 4.1-4.4.

This work extends the implementation presented in section 4.3: "Variations on a Scheme – Nondeterministic Computing." The section uses implicit continuations to implement the `amb` keyword. This work extends that in two ways by:

- providing an arbitrary number of continuations, rather than only two (success and fail); and
- exposing continuations as first-class objects using the `call/cc` interface

Additionally, a significant portion of this project was dedicated to exploring the theory and use cases of continuations, so there will be some backgrounds and examples of common use cases.

2 Background

Simply due to curiosity, this project became largely exploratory in nature. This project ultimately aims to answer the question: "What are continuations and when should we use them?" Thus this background section forms the bulk of the report and the research effort, despite the original goal to implement a new feature.

From Matt Might's blog post Continuations by example: Exceptions, time-traveling search, generators, threads, and coroutines:

> Continuations are the least understood of all control-flow constructs. This lack of understanding (or awareness) is unfortunate, given that

> continuations permit the programmer to implement powerful language features and algorithms, including exceptions, backtracking search, threads, generators and coroutines.
>
> I think part of the problem with continuations is that they're always explained with quasi-metaphysical phrases: "time travel," "parallel universes," "the future of the computation." I wrote this article so that my advanced compilers students could piece together how continuations worked by example.

I wholly agree with this sentiment. There are a dozen different ways to interpret continuations, and each interpretation lends its own insights. In this section I will attempt to give a better idea of what continuations are, what they are capable of, how they are used, and even how they may be implemented – all this just to get an inkling of what this mysterious control structure is.

2.1 What are continuations?

As Might points out, a common way to describe continuations are that they are the "future" of an expression. Luckily, Scheme's syntax is able to provide an objective way to view the future of an expression: the future of an expression is the containing expression. This is due to the fact that parameters are fully evaluated before the function is applied. Consider Figure 1. The result of the subexpression `(* a b)` is the addition by `c`, so we can express its continuation as a procedure that performs this action: `(lambda (val) (+ val c))`. The futures (continuations) for each subexpression are also shown in the figure.

Another way to think about it is that the continuation for any expression is the (implicit) callback for a given computation. After `(* a b)` is fully computed, then it should send its result to the lambda function representing its continuation. This should not feel too strange to those familiar with asynchronous programming: often the result of an asynchronous computation is passed to a specified callback or handler function.

Continuations become very useful if the programmer has access to them. For example, since the `(* a b)` is an ordinary procedure, we can save this procedure to a variable and call it like an ordinary function. If the programmer saves the continuation of that subexpression to a variable `cont`, then it can be invoked. `(cont 42)` with `5` assigned to `c` will produce `47`.

Of course, this is not a complete characterization, and the rest of the background section will be used to foster a more complete understanding of continuations. At many times, continuations will feel like an analogous construct in another language; it may seem like a "first-class return" or a "snapshot of the program state"; no single characterization is universally better than another.

Note that each expression or statement (in any programming language) has an implicit continuation, and continuations are an incredibly general concept. For example, the continuation of a statement in a sequence of statements is the next statement. However, we are only concerned with *reified* continuations in

```
;;; consider the following expression: a*b+c
(+ (* a b) c)

;;; future of a:         (lambda (val) (* val b))
;;; future of b:         (lambda (val) (* a val))
;;; future of (* a b):   (lambda (val) (+ val c))
;;; future of c:         (lambda (val) (+ (* a b) val))
```

Figure 1: Continuation as the future of an expression

this report (i.e., continuations with a concrete implementation in the programming language), and often *first-class* reified continuations (continuations that appear as first-class objects to the programmer).

2.1.1 The `call/cc` API

In the previous section, continuations are characterized as procedures; but how do we get these procedures? In Scheme, implicit continuations are exposed to the programmer using the `call-with-current-continuation` procedure, which is often aliased to `call/cc`. `call/cc` takes as argument a procedure that takes the current continuation as a parameter. The continuation is a first-class object – it can be stored in variables, passed as a parameter to functions, returned from functions, and be a member of complex data structures.

`call/cc` will then execute the procedure. If the continuation is not invoked, then the return value of the procedure is passed to the continuation of the `call/cc` expression. If and when the continuation is invoked, then the value given during the invocation is passed to the continuation of the `call/cc` expression. Several of `call/cc` usage are shown in Figure 2.

2.1.2 Continuation-passing style (CPS)

Continuation-passing style is a very instructive tool for discussing continuations at a functional level.

First-class continuations can be implemented in any language with lambdas (functions with lexical closures), by performing a syntactical transfomation of functions to CPS, which is shown in Figure 3. Looking at the CPS form, we can characterize a continuation in the following ways:

- Each function takes its continuation as an extra parameter.
- A continuation is only invoked as a tail-call (if a continuation is called, its “then-current continuation will be abandoned”, so a non-tail call would be wasteful).
- A continuation takes the result of the previous instruction as its singular parameter.

```
;; the original expression
(+ 1 (+ 2 (+ 3)))
;; => 6

;; continuations are optionally-resumable:
;; i.e., may not be invoked at all
(+ 1 (call/cc (lambda (cc) (+ 2 (+ 3)))))
;; => 6

;; invoking a continuation causes the current
;; continuation to be discarded; the continuation
;; of `(cc 3)` is `(lambda (val) (+ 2 val))`, and
;; this is never invoked
(+ 1 (call/cc (lambda (cc) (+ 2 (cc 3)))))
;; => 4

;; continuations are multiply-resumable:
;; i.e., may be invoked multiple times
(define cont '())
(+ 1 (call/cc (lambda (cc) (set! cont cc) (+ 2 (+ 3)))))
(cont 4) ;; => 5
(cont 5) ;; => 6
```

Figure 2: Sample usage of call/cc

```
;;; without continuations; uses regular return
(define (add a b)
  (+ a b))
(define (mult a b)
  (* a b))
(define (a*b+c a b c)
  (add (mult a b) c))

(display (a*b+c 1 2 3)) ;; => 5

;;; continuations using CPS
(define (add a b cont)
  (cont (+ a b)))
(define (mult a b cont)
  (cont (* a b)))
(define (a*b+c a b c cont)
  (mult a b (lambda (a*b)
              (add a*b c cont))))

(a*b+c 1 2 3 display) ;; => 5
```

Figure 3: Continuation passing style example

- A function written using CPS *never returns normally* – it must exit by (tail-)calling a continuation.

Note that continuations in CPS are ordinary functions ("callbacks" in event-driven coding), and thus can be used to implement continuations in a language that doesn't support implicit continuations. CPS is used by Abelson and Sussman to implement continuations in their nondeterministic interpreter, and this is extended for this project.

Other ways to implement continuations will be discussed in §2.5.2.

2.2 Nondeterministic computing and the `amb` keyword

Abelson and Sussman implement the `amb` keyword in Scheme. This keyword takes a set of possible values as input, and produces a value[1] that satisfies all of the assertions placed on the value in the future. For example, the example shown in Figure 4 may assign to `a` either `3` or `5`.

This can be implemented using a search over all the possible solutions. What is really impressive is that we are able to state SAT problem in a declarative

[1]It is "nondeterministic" because the keyword only has to return *a* value that satisfies the assertions (conditions). The exact choice of value is not important. This is a SAT-solver, as Matt Might implements using CPS in his blog post on continuations.

```
(define a (amb '(1 2 3 4 5)))
(require a odd?)
(require a prime?)
```

Figure 4: Sample usage of `amb`

manner, rather than embedding it in an algorithm[2]. Even more impressively, we are able to state the constraints on an ambiguous value in its *future*, and possibly after the value has already been used.

The key part to this implementation is the implicit[3] use of continuations. In this case, an additional error continuation is provided to handle the case of a condition failing – this causes the next value to be tried (if any). This also necessarily undoes any side effects (such as variable mutation) to effective "time travel" to the point in the program where the ambiguous value is declared.

Functionally, this can be thought of as a try/catch statement, without the user having to explicitly code the control flow (and simply because there is not a builtin try/catch statement in Scheme).

The interpreter from section 4.3 is based off of the version in 4.1, which uses a procedure-based intermediate representation (IR) for all expressions. The expression is first parsed using the Scheme `read` procedure, and then semi-compiled into this IR to avoid parsing every time the expression is encountered (if the expression is invoked multiple times).

The form of the procedure representing the compiled expression (the expression IR) is shown in Figure 5. Whenever the expression is invoked, this lambda is called with the current runtime environment.

The analogous expression IR for the nondeterministic backtracking interpreter from section 4.3 is shown in Figure 6. At runtime, each expression is also given two continuations as parameters – this is known as continuation-passing style (CPS). These continuations are ordinary procedures that should be used to pass around the results of computations. The success continuation takes the value from the previous computation and the failure continuation. The failure computation is special, its only purpose being to discard the value and reverse the stack and side effects so that the next value can be tried – it does not need the value from the previous computation nor the success continuation.

[2]The same is true for generators, and is what makes them so powerful.

[3]This is my own terminology. I use "implicit" to refer to (reified) continuations that are baked into the language and can be used by the interpreter to perform control flow. Optionally, the interpreter may choose to expose these continuations via interfaces such as `call/cc`. I use "explicit" to refer to CPS, in which reified continuations are implemented by the user as explicit procedure calls.

```
(lambda (env)
  ;; env stores the symbol table for the current scope
  ...)
```

Figure 5: Expression IR from section 4.1

```
(lambda (env succeed-cont fail-cont)
  ;; succeed-cont is of the form (lambda (value fail-cont) ...)
  ;; fail-cont is of the form (lambda () ...)
  ...)
```

Figure 6: Expression IR from section 4.3

2.3 Continuations versus other control-flow constructs

This section is structured similarly to the Wikipedia page on coroutines, which is an excellent resource on coroutines.

2.3.1 Continuations vs. gotos

Gotos are the software equivalent to a jump instruction. They are very simple and limited. For example, they can only jump to other locations in the current function; jumping to another function without modifying the stack would cause relative-addressing (local variables) to break. Also unlike gotos, continuations usually return a value.

Continuations are more similar to C's `setjmp` and `longjmp` functions, which essentially save and restore a history of the stack. Not only does this allow you to jump to different functions, it also allows you to jump back multiple levels of the stack. However, reified continuations are more powerful (and expensive) in that they are both optionally-resumable and multiply-resumable[4].

In his blog post about continuations, Matt Might suggests an idiom that behaves very similarly to `setjmp`/`longjmp`. For those who are unfamiliar with these functions, `setjmp` is similar to `fork` in that returns different values depending on the context. `setjmp` is encountered either it is the next statement to execute, or when `longjmp` jumps to it. In the first case, it will return a falsy value, and in the latter it will return the value specified in the `longjmp` call.

Might does not explicitly mention `setjmp`/`longjmp` in his blog post, but the interface is remarkably similar. See a simple error-handling program with a nonlocal jump in C/C++ is shown in Figure 7. The equivalent program in Scheme, using the `setjmp` idiom (Might calls this `current-continuation` in his examples), is shown in Figure 8.

[4]In order to be multiply-resumable, a continuation essentially needs to create a copy of the stack whenever it is executed. See the section on implementation

```
main() {
  jmp_buf env;
  int val;

  val = setjmp (env);
  if (val) {
    fprintf (stderr,"Error %d happened",val);
    exit (val);
  }

  /* code here */

  longjmp (env,101);   /* signaling an error */
}
```

Figure 7: Sample usage of `setjmp`/`longjmp`. Source: cplusplus.com: `setjmp`

Note that both `goto` and `setjmp` may be dangerous if there are side effects in the jumped code. `goto` is very dangerous since it can actually branch forward in a function, skipping variable initializations completely. `setjmp` is less dangerous in that it can only go back to a previous location on the stack (which should be safe), but may cause issues with dangling or incorrect pointers if cleanup code is skipped. Continuations (in Scheme) are safer because they cannot branch forward in time, and because garbage collection would manage dangling references.

2.3.2 Continuations vs. return

A continuation is like a return statement in that it usually passes a value and it (conceptually) unwinds the stack to a given location. However, continuations are optionally-resumable, multiply-resumable, and may unwind the stack more than one stack frame[5].

2.3.3 Continuations vs. callbacks

In CPS, continuations *are* simply callbacks that follow a specific form. In general, first-class continuations feel very much like ordinary callbacks, in that they are invoked with the result of a computation. For example, in Javascript where event-driven coding is common, callbacks to asynchronous functions and the `Promise` API look very similar to CPS.

[5]The latter is useful in exception handling, or when needing to jump back many stack frames without the overhead of returning from each stack frame, such as in the backtracking nondeterministic interpreter.

```
(define (setjmp)
  (call/cc (lambda (cc) (cc cc))))

(let ([longjmp (setjmp)])
  (if [procedure? longjmp]
      ;; val == false
      (begin
        ;; code here
        (longjmp 101))
      ;; val = longjmp; val != false
      (error 'setjmp-example
             "Received error signal from longjmp"
             longjmp)))
```

Figure 8: Useful Scheme idiom using continuation that behaves like setjmp. Source: Continuations by example: Exceptions, time-traveling search, generators, threads, and coroutines

Of course, in the case of implicit continuations, there may be special machinery that manages program execution state that cannot be performed with a regular procedure call, but the API is the same as a procedure.

Note that callbacks are also multiply-resumable, optionally-resumable, and capture program state (capture its lexical environment). In general, we can think of continuations as a specific type of callback.

2.3.4 Continuations vs. monads

Haskell users may know that continuations exist as a monad (`Control.Monad.Cont`). Even for those who don't (but are familiar with monads), the continuation's structure should feel monadic:

- Continuations act like a wrapper around a computation.
- Chaining computations looks very similar to the bind operation.
- Continuations are a control-flow structure.

Consider Figure 9, the Haskell analog of the CPS example shown in Figure 3. Note that Haskell also has the `callCC` interface, similar to the `call/cc` procedure. Figure 10 shows a sample monad instance declaration for the continuation type. Unsurprisingly, the syntax between Scheme and Haskell turns out to be very similar, even when Scheme continuations feel like procedures and Haskell continuations feel like monads.

```haskell
import           Control.Monad.Cont

mult             :: Int -> Int -> Cont r Int
mult a b         = return (a * b)

add              :: Int -> Int -> Cont r Int
add a b          = return (a + b)

-- a * b + c
atbpc            :: Int -> Int -> Int -> Cont r Int
atbpc a b c      = mult a b >>= add c

main = do runCont (atbpc 1 2 3) print
```

Figure 9: CPS in Haskell using `Control.Monad.Cont`

```haskell
newtype Cont r t = Cont ((t -> r) -> r)
  -- a value of type Cont r t is of the form Cont f,
  -- where f is a function of type (t -> r) -> r

runCont (Cont f) q = f q

instance Monad Cont where
  return :: t -> Cont r t
    -- return takes a value of type t and produces a Cont
    -- computation of type t with response type r.
  return x = Cont (\q -> q x)
    -- produce a Cont computation which given a question q,
    -- applies the question to the value x.

  (>>=) :: Cont r t -> (t -> Cont r s) -> Cont r s
  x >>= f = Cont (\q -> runCont x (\v -> runCont (f v) q))
```

Figure 10: A simplified continuation monad in Haskell. Source: Cont computations as question-answering boxes

2.3.5 Continuations vs. threads

Continuations are like dormant, multiply-resumable threads. In particular, threads represent the current execution context of a program, and can be thought of as a snapshot of location on the stack. However, you cannot resume a thread from the same position multiple times, unlike continuations.

WikiWikiWeb's ContinuationExplanation hints that threads can be used to implement continuations, and vice versa. It also mentions that some thread implementations, such as POSIX pthreads, are not as powerful as continuations and cannot fully implement all continuation use cases. Continuation implementation will be further discussed in §2.5.2.

Continuations and execution context (and in particular, concurrent threads of execution) is part of the recurring theme, and will appear again in the discussion of coroutines.

2.4 Uses for continuations

2.4.1 Callbacks and promises

Even in languages without implicit continuations, callbacks (that look and act a lot like continuations) are widely known and applied. Might's blog post By example: Continuation-passing style provides the example of a CPS-style wrapper around the legacy `XMLHttpRequest` Ajax API[6] that may look extremely natural for Javascript users, even without knowing about continuations.

Another asynchronous programming model in Javascript is the `Promise` API, which can be thought of as a modern wrapper around callbacks with some nice features. Promises are objects that contain a computation, and allow you to specify a success and error continuation explicitly using the `Promise::then()` and `Promise::catch()` methods. Javascript promises have a number of interesting features that make them more appealing than ordinary callbacks, such as their ability to be `await`-ed to avoid "callback hell," the ability to automatically chain continuations, and the ability to attach multiple continuations to the same operation (which would be called asynchronously due to Javascript's event loop).

Since promises and callbacks are more or less functionally equivalent, scenarios that are expressible using callbacks are also expressible using promises. If we turn the example from Figure 3 into CPS, and then transform that to use the `Promise` API, then we would get Figure 11.

As stated earlier, continuations model only a specific subset of callbacks or promises that are useful for execution; namely a single success callback that takes a single parameter (the result of the previous operation), and (optionally) an error callback that takes a single parameter (the error message). For my project, I aim to generalize implicit continuations to the case where there are

[6] I'm not sure when this blog post was written, since the modern `fetch` API uses promises rather than callbacks.

```
add = (a, b) =>
  new Promise((succeed_cont, fail_cont) => succeed_cont(a+b));

mult = (a, b) =>
  new Promise((succeed_cont, fail_cont) => succeed_cont(a*b));

atbpc = (a, b, c) =>
  new Promise((succeed_cont, fail_cont) =>
    mult(a, b).then(apb =>
      add(apb, c).then(succeed_cont)));

atbpc(1, 2, 3).then(console.log);
```

Figure 11: CPS using Javascript `Promise`s

an arbitrary number of continuations, which is similar to how a function is not limited in the number of callbacks it receives.

2.4.2 Nonlocal branching

An example of nonlocal branching is the nondeterministic interpreter in *SICP* section 4.3. Whenever there is a requirement conflict, we have to unwind the stack to the location of the relevant `amb` invocation, which may be nonlocal.

A personal example I have is when I had to write an A*-search algorithm and chose Scheme. Iterative deepening with a time limit was used; when the time limit was reached, rather than unwinding up the arbitrarily-deep search stack, a continuation served as a much simpler nonlocal goto. (Continuations were not used for the DFS.)

2.4.3 Exception handling

This can be thought of as an example of nonlocal branching. To implement a try/catch block, a stack of error (failure) continuations has to be implemented alongside the ordinary (success) continuation. Matt Might shows this in his blog post about continuations.

Alternatively, rather than only passing the ordinary continuation to each function in CPS, both the success and failure continuations may be passed as parameters to every function. This may be a little bit more inefficient but has the same effect. This is how failure continuations were passed around in the nondeterministic interpreter.

2.4.4 Coroutines (and generators)

As mentioned earlier, continuations are functionally very similar to (and potentially more powerful than) threads. Thus, continuations may be used to implement cooperatively-scheduled threads[7], coroutines (equivalent to cooperative threads), and generators (a subset of coroutines).

Cooperative threading system Continuations can be used to implement a cooperatively-scheduled threading system by maintaining a collection of continuations representing threads. Each thread may voluntarily yield to another available thread via the yield instruction. The collection may be a queue and the next continuation in the queue will be chosen when a thread yields; this forms a very simple round-robin scheduling algorithm. Matt Might's blog post on continuations includes this as an example. This allows us to create concurrency (but not parallelism) in our interpreter.

Kotlin has an implementation of cooperative threading that is called coroutines (which are equivalent – see next example).

Coroutines A coroutine is a generalization of a subroutine (procedure) that maintains execution state between calls. It may exit either normally (by returning) or by *yield*ing to another coroutine; when it is invoked again, it will resume execution from the yield point. Coroutines are functionally equivalent to cooperatively-scheduled threads.

A simple way to implement coroutines is to store a table of continuations for each function. Yielding to a coroutine would mean looking up its continuation and invoking it with the yielded value.

Generators Generators are a restricted type of coroutine that is very useful. A generator is a function that yields values one at a time. They are useful as iterators over arbitrary data structures. To the outsider, calling the function successively produces successive values of the collection; this works because the function saves its execution context between calls and thus doesn't lose its position when iterating over the data structure.

Generators offer a nice solution to iteration that are roughly equivalent to streams (streams are employed in *SICP* section 4.4 as an iterator, but a generator would have worked as well). If neither streams nor generators were present, then implementing an iterator function without nonlocal state becomes difficult.

Generators are a strict subset of coroutines because the generator function can only yield to its caller, while a full coroutine can yield to any other coroutine. This simplifies the implementation and does not require a table of continuations. Rather, the two functions (generator and consumer) simply pass back and forth the new continuations and the yielded values.

Many languages implement generators due to their ubiquity, even if they do not implement coroutines.

[7]Threads are typically *preemptively scheduled*, which means that they yield control to another thread at arbitrary moments in their execution (generally to maintain performance or responsiveness). *Non-preemptive* or *cooperative* scheduling means that a thread only yields to another thread explicitly.

Communicating sequential processes (CSP) CSP (not to be confused with CPS) is a formal description of communication between concurrent threads of execution. Coroutines (i.e., cooperative threads) are generally the mechanism associated with CSP to allow communication between processes. A coroutine can efficiently yield its execution to a different coroutine; a normal thread complicates matters with its preemptive scheduling.

An example of this is Kotlin's coroutines. Go's goroutines are somewhat of a hybrid between the two: they are lightweight user-space threads that normally are preemptively scheduled, but have the ability to yield execution voluntarily. Go has a mechanism for communication between goroutines called *channels* that are intended to model CSP.

2.4.5 Compiling with continuations

Continuations, and CPS in particular, have been used as a tool for expressing control flow as a compiler IR, usually alongside TCO[8]. This dates back to the first Scheme compiler, Rabbit, which used CPS as an IR. Since continuations replace return statements, the stack model may be replaced with a similarly efficient model involving only continuations. Tail calls can also be thought of as "gotos with parameters," which lends itself to efficient implementation.

Two resources on this are Matt Might's CPS Conversion and Andrew Appel's *Compiling with Continuations*[9].

2.5 Miscellaneous notes about continuations

2.5.1 One procedure call, multiple "returns"

One property of continuations made clear by the comparison to `setjmp` with the idiom in Figure 8 is that the `call/cc` interface looks a lot like an ordinary function call to the caller, in that it will (typically[10]) return a value to the caller.

However, `call/cc` also creates a branch point (think goto label), which means that multiple values may be "returned" from that location if the continuation is invoked multiple times (recall that continuations are multiply- and optionally-resumable, just like an ordinary goto label).

Note that this does not mean that a function may return by calling its current continuation multiple times sequentially; the current continuation is discarded and replaced by the the invoked continuation. This is illustrated in Figure 12. The current continuation is called with value 1, and the current continuation of invoking the continuation (which is the next statement) is discarded, so control never reaches the next statement. This is equivalent to saying that invoking a continuation is only useful as a tail call, since any instructions that come after (i.e., the continuation of the invoking the continuation) are discarded.

[8]Note that CPS can be used without TCO, such as in the Chicken Scheme compiler, since it is transpiled to C (in which implementing TCO may add considerable complexity), and thus requires other optimizations to clean up the stack.

[9]I haven't had the time to read these, but the latter is a famous text on the matter.

[10]Unless a continuation is invoked that unrolls the stack further than the calling statement.

```
(define (does-not-multiply-return)
  (call/cc (lambda (cc)
             (cc 1)     ;; expression a
             (cc 2))))  ;; expression b

;;; The continuation of expression a is expression b.
;;; Invoking the continuation `cc` discards the
;;; continuation of expression a, i.e., expression b
;;; will not be executed.

(does-not-multiply-return) ;; => 1
```

Figure 12: The current continuation gets replaced by a call to a continuation

Another way to think of this is that you cannot call `return` in C multiple times sequentially, as the second one is unreachable; however, what you can do is save the `return` statement as a first-class object and (at some later point in reachable code) invoke that statement to have the same behavior as returning from that function.

2.5.2 Efficient implementation

It's been hinted several times that the implementation of continuations would be similar to the implementation of threads, since both involve saving a snapshot of the execution state. Since continuations are multiply-resumable, it is more difficult to implement efficiently than a thread, since we need to maintain a pristine copy of the stack in case it is invoked more than once.

Another issue that makes it harder to optimize is that continuations have *unlimited extent* – they exist over the lifetime of the program, potentially outside of the context (closure) that they were created in. Thus it is possible to not only unroll the stack to a previous position, but to also enter a deeper position in the stack. As a result, optimizations must maintain a reference to all parts of the stack that are present at the time of the continuation being called.

Segmented stacks and *cactus stacks* are optimizations that allow multiply-resumable continuations unlimited extent by branching the stack and reusing memory when possible. Copy-on-write (COW) is a common memory optimization technique that also applies here to avoid copying the stack unless necessary. A more complete discussion of implementation is given in WikiWikiWeb: ContinuationImplementation.

As mentioned previously, one way to implement continuations is using CPS. CPS usually depends on tail-call optimization (TCO).

We can simplify the implementation by restricting the capability of continuations. One possible restriction is to only allow singly-resumable (but still optionally-resumable) continuations. Another restriction is to only allow up-calls (unwinding the stack, i.e., limited extent) – these are called *escape continuations.*

2.5.3 Flavors of continuations

In the previous section, we have already talked about two forms of restricted continuations: single-use and escape continuations. Continuations with both of these restrictions of these are very similar to `setjmp` – this is still powerful enough for exception handling, generic nonlocal jumps. The WikiWikiWeb article on single-use continuations notes that `setjmp` can be used to implement a multi-use branch point using the very simple idiom `while(setjmp(buf));`. The Wikipedia article on continuations notes that escape continuations are a means to implement TCO.

There are also *delimited continuations*, which are a generalization of the continuations mentioned thus far. Rather than capturing the entire execution state, a part of the stack can be captured. Its API are the `shift` and `control` procedures (rather than `call/cc`).

2.6 Analogs in other languages

Several examples of continuations in other languages have been provided; this section serves as a summary.

In C, the flow-control capability of `goto`, `return`, and `setjmp` are all a subset of the ability of continuations. Of these, `setjmp` is the most powerful and are roughly equivalent to single-use escape continuations.

In Haskell, continuations are exposed as a monadic type. Expressing chained computations using the bind (>>=) operator is very natural.

Many languages have some support for generators (e.g., Python), cooperative scheduling (e.g., Kotlin's coroutines), and CSP (e.g., Golang's Goroutines). These may not explicitly be implemented using continuations, but the machinery is likely similar.

3 Multiple continuations

For this project, I decided to extend the Scheme interpreter by implementing a system of *multiple continuations* in programmer-space via a `call/cc`-like interface.

The following terminology should be established since we are working with Scheme code in the implementation of the interpreter, as well as Scheme code that may be run in the interpreter. This is non-standard – I'm not sure what the convention is for interpreter-writers.

Programmer-space This refers to code that the programmer may type into the interpreter.

Interpreter-space This refers to code in the implementation of the interpreter (including any IRs for programmer-space constructs).

3.1 Functional API

Functionally, multiple continuations is equivalent to CPS with an arbitrary number of continuations (callbacks), but the continuations are defined implicitly so that the programmer does not have to carry the continuations through all of the function signatures. This is a simple generalization of the single-continuation scheme and may be useful where multiple "futures" are useful, such as a default and error future in exception handling.

The implementation is based on sections 4.1 and 4.3 of *SICP*. The nondeterministic interpreter uses a CPS IR but doesn't expose continuations to the user. It implements two continuations for every expression: a default (success) continuation, and a special-purpose backtracking (failure) continuation[11]. For this project, the special failure continuation is removed, and the default continuation is replaced with a list of continuations.

To the programmer, the introduction of multiple continuations change would look like the creation of a few APIs, described below, that are not present in the interpreter given in the book. The first is the familiar `call/cc` API (which does not require multiple continuations). In the case of the latter APIs, we establish the convention that the first continuation is the default continuation, the second continuation is an error continuation, and any others are user-defined.

3.1.1 Exposing continuations via the `call/cc` API

This API should appear to the programmer exactly like the builtin `call/cc` implementation described in §2.1.1. In particular, it should call the provided procedure with the current (default) continuation as the sole argument. The continuation should be a first-class object, and invoking it should appear no differently than an ordinary function call. If a continuation is invoked, then the current continuation should be discarded.

This is implemented in terms of the more general case for multiple continuations; the implementation is shown in Figure 13.

3.1.2 Multiple continuations via the `call/ccs` API

This API is very similar to `call/cc`, except that the argument to the function is the list of current continuations.

As an example, the code to send a value to the error continuation (to `throw` a value) is shown in Figure 13.

3.1.3 User-defined continuations via the `call/new-ccs` API

This API is unlike anything in the Scheme language, so it is worth describing it here. This procedure takes two arguments: a procedure that transforms the

[11]Alternatively, the failure continuation may be implemented in an external runtime stack rather than be passed around to every function as a parameter – this may be more efficient. Exceptions may also be handled this way.

```
;;; implement `call/cc` using `call/ccs`
(define (call/cc f)
  (call/ccs (lambda (ccs)
              (f (car ccs)))))

;;; implement throw using `call/ccs`
(define (throw val)
  (call/ccs
   (lambda (ccs)
     ((cadr ccs) val))))

;;; swap the default and error continuations using `call/new-ccs`
(call/new-ccs
  ;; cc transformer function swaps first and second ccs
  (lambda (ccs)
    (cons (cadr ccs)
          (cons (car ccs)
                (cddr ccs))))

  42             ;; goes to error continuation
  ; (throw 42)   ;; goes to default continuation
)
```

Figure 13: Intuitive of the API functions

current continuations and produces the new continuations, and an expression to invoke with the new ccs.

As an example, the code to switch the default and secondary continuations is shown in Figure 13.

3.2 Implementation

All of the expression syntactical analysis functions in the IR must return a CPS function, in the same manner as the nondeterministic interpreter. However, instead of receiving two continuations, it receives a list of continuations. For almost all cases, the procedure is exited by calling the first (default) continuation with the expression value.

The tricky part is that every time we invoke an IR procedure of this form, we must supply an array of continuations. If we need to modify the default continuation, then we must `cons` this to the rest of the original continuations. This makes the CPS more tedious.

To represent the continuations (ordinary procedures in interpreter-space) as procedurs in programmer-space, we must wrap them with some metadata like we do for primitive and compound procedures. In this case, we simply tag the continuation with the symbol `'continuation`. Continuation invocation is treated the same as a primitive procedure in `execute-application`, except that the current continuations are discarded when a continuation is executed.

The implementations for `call/ccs` and `call/new-ccs` is shown in Figure 14. The implementation for `call/cc` was already shown above; it is a special case of `call/ccs`, and can be trivially written in terms of it. (Note that while the former two are *special forms*, the latter is an ordinary procedure in programmer-space.)

Note that the continuations for the user-specified continuations in `call/new-ccs` are the continuations for the `call/new-ccs` expression. This essentially automatically creates a stack of continuations. For example, throwing an exception inside a try/catch block would go to its error continuation (the catch clause). Throwing an error inside the catch clause would then go to its error continuation (which should be the error continuation of the try/catch block).

I should also add that I do not know how continuations and the `call/cc` interface are implemented in Scheme (and the implementation may differ between Schemes). This is also not intended to be a very efficient implementation, but is rather a proof-of-concept that continuations can be implemented very simply in a functional language with lambda closures, garbage collection, and TCO.

There may be a number of safety issues with `call/new-ccs`, especially with the ability to overwrite the default continuation. For this exploratory assignment, I have left it alone in all its dangerous glory, but this should probably not be productionized.

3.3 Examples

Implementing try/catch is very simple in programmer-space, and is shown in Figure15 (the implementation for `throw` was already shown in Figure 13). The only caveat is that the try and catch clauses must be provided as lambdas rather than plain expressions, in order to avoid their eager evaluation. (If it were implemented as a special form, or if syntactic macros were implemented in the interpreter, plain expressions would be possible.)

Matt Might also provides an example of generators using continuations in his blog post Continuations by example: Exceptions, time-traveling search, generators, threads, and coroutines. While this does not use multiple continuations at all, it is a good proof-of-concept of generators, and is more evidence that my implementation of `call/cc` is correct. My implementation of the generator in programmer-space is not the most convenient form. Might writes his in terms of syntactic macros to make the syntax much more appealing. My implementation is shown in Figure 16.

3.4 Source code

My implementation as the files `4.1/4.1.scm` and `cont/conts.scm` in the GitHub repository jlam55555/sicp. Chez Scheme 9.4 was used as the Scheme of choice.

```
(define (mi::analyze-call/ccs exp)
  ;; for (call/cc f), call f, binding the continuation
  ;; as a primitive procedure to the argument of f
  (let ([fproc (mi::analyze (mi::call/ccs-arg exp))])
    (lambda (env ccs)
      (fproc
       env
       (cons
        (lambda (proc)
          (mi::execute-application
           proc
           (list
            (map (lambda (cc) (list 'continuation cc)) ccs))
           ccs))
        (cdr ccs))))))

(define (mi::analyze-call/new-ccs exp)
  (let ([fproc (mi::analyze (mi::call/new-ccs-generator exp))]
        [body (mi::analyze (mi::call/new-ccs-body exp))])
    (lambda (env ccs)
      (fproc
       env
       (cons
        (lambda (proc)
          (mi::execute-application
           proc
           (list
            (map (lambda (cc) (list 'continuation cc)) ccs))
           (cons
            (lambda (new-ccs)
              ;; no safety checks!!! run body with
              ;; user-supplied ccs mapped to primitive procs
              (body env
                    (map
                     (lambda (cc)
                       (lambda (val)
                         (mi::execute-application
                          cc
                          (list val)
                          ccs)))
                     new-ccs)))
            (cdr ccs))))
        (cdr ccs))))))
```

Figure 14: Implementations of call/ccs and call/new-ccs

```
(define (try/catch try catch)
   ;; try is a thunk
   ;; catch is a proc that takes the error msg as input
   ;; both try and catch return a value
   (call/new-ccs
    (lambda (ccs)
      (cons (car ccs)
        (cons catch
              (cdr (cdr ccs)))))
    (try)))

(define (my-/ x y)
  ;; modified version of `/` that calls the error
  ;; continuation on div0
  (if [= y 0]
      (throw "my-/: /0")
      (/ x y)))

(try/catch
 (lambda () (my-/ 1 2))
 (lambda (err) "oh no"))
;; => 1/2

(try/catch
 (lambda () (my-/ 1 0))
 (lambda (err) "oh no"))
;; => "oh no"

(try/catch
 (lambda () (throw 32))
 (lambda (err) (throw 54) "oh no"))
;; => "top-level error continuation called with irritant 54"
```

Figure 15: `try/catch` implementation and usage

```scheme
(define (current-continuation)
  (call/cc (lambda (cc) (cc cc))))

(define (tree-iterator tree)
  (lambda (yield)
    (define (walk tree)
      (if [not (pair? tree)]
          (yield tree)
          (begin
            (walk (car tree))
            (walk (cdr tree)))))
    (walk tree)))

(define (make-yield for-cc)
  (lambda (value)
    ;; `yield` implementation
    (define cc (current-continuation))
    (if [procedure? cc]
        ;; when called from generator, return to for loop cont
        (for-cc (cons cc value))
        ;; when called from for loop, return to generator cont
        (void))))

(define (for-generator iterator body)
  (define (loop iterator-cont)
    (define cc (current-continuation))
    (if [procedure? cc]
        ;; get next value using the generator continuation
        (if iterator-cont
            (iterator-cont (void))
            (iterator (make-yield cc)))
        ;; value handler: receive new value and continuation
        [begin
          (body (cdr cc))       ; next generator value
          (loop (car cc))]))    ; next generator continuation
  ;; begin iterator loop with no iterator continuation
  (loop false))

(define (println val) (display val) (display "\n"))

(for-generator (tree-iterator (cons 3 (cons (cons 4 5) 6)))
               println)
;; => 3 4 5 6
```

Figure 16: Demonstration of generators using `call/cc`

4 Conclusion

I was able to explore continuations to a much greater depth of understanding than before. Perhaps more importantly, this discussion has piqued my interest on a number of other related topics, including: segmented and cactus stacks; delimited continuations; implementation of Javascript `Promise`s and their relationship to monads; goroutine implementation and relationship to coroutines; reading Andrew Appel's *Compiling with Continuations*.

Unfortunately, my extension to the continuations API is probably not that useful; with the exception of the simple implementation of try/catch in programmer-space, I could not think of any other use cases where this API is more useful than simply passing around multiple continuations (or simply callbacks) as needed. Efficiency was not a concern during this project, but this new feature probably adds a nontrivial overhead to function calls as well.

5 References

Since most of the sources are not academic with auto-generated BibTeX citations, and are easily accessible via the web, I have chosen to include the sources as links throughout the document in lieu of a more formal citation style.