

# Handout I: Continuations

Christoph Reichenbach

December 9, 2021

Perhaps the first abstraction that programmers learn are *variables*, constructions that can bind to values (i.e., the results of computations). We can use variables in most places that expect a literal value or an expression<sup>1</sup>, including the middle of computations such as `PRINT x + 1`.

*Continuations* allow us to abstract over the opposite of values: they allow us to abstract over the context in which values get used. For example, consider the following MYSTERY program fragment:

```
1  VAR x : INT;  
2  VAR y : INT  
3  BEGIN  
4    ...  
5    PRINT x + 1;  
6    PRINT y  
7  END
```

There are several continuations in this program. Let us consider first the continuation in line 6, at variable `y`. Since a continuation is the *opposite abstraction* of a variable, this means that the continuation at variable `y` is the code that uses `y` and then continues — in this case, the continuation would be ‘take an integer value, PRINT that value, and then END the program’.

Another continuation would be in line 5. This second continuation would be ‘take an integer value, add 1 to it, PRINT the result, then PRINT `y`, and finally END the program’.

Continuations were originally a theoretical concept for modelling jumps and loops in programs, and, later, for modelling exceptions and cooperative forms of concurrency. Today, they are first-class concepts in Scheme, Racket, Ruby, Haskell (albeit only in bounded form), and (as language extensions) the two main implementations of Standard ML, SML/NJ and Mlton.

In SML, we can access continuations by opening the following module:

```
open SMLofNJ.Cont
```

This imports a number of definitions, most prominently:

```
...  
type 'a cont  
val callcc : ('a cont -> 'a) -> 'a  
val throw : 'a cont -> 'a -> 'b  
...
```

Here, the type  `$\alpha$  cont` describes a continuation for a value of type  `$\alpha$` . In our introductory example,  `$\alpha$`  would be `INTEGER`. The two main operations on continuations are:

- `callcc`, which is short for ‘call with current continuation’. `callcc` takes a single argument `f` of type  `$\alpha$  cont  $\rightarrow \alpha$` . Here, `f` is a function that receives the *current continuation*, meaning the continuation `c` of what happens to the return value of `callcc (f)`. We can return from `callcc (f)` in two ways:

---

<sup>1</sup>There are a few limitations in practice, such as e.g. C++’ `const_expr` or Ada and Mystery’s subrange type bounds.

1. By evaluating  $f(c)$ , where  $c$  is the ‘current continuation’ around `callcc (f)`. In this case, the return value of  $f(c)$  also becomes the return value of `callcc(f)`.
  2. After starting to evaluate  $f(c)$ , by *throwing* the current continuation  $c$  from any point in the program. To do so, we rely on the second operation:
- `throw`, which *throws* a continuation (analogously to raising an exception). `throw` has the type

$$\text{throw} : \alpha \text{ cont} \rightarrow \alpha \rightarrow \beta$$

meaning that it takes two parameters: a continuation that expects a value of type  $\alpha$ , and a value of type  $\alpha$  to plug into that continuation. `throw` claims to return type  $\beta$ ; however, `throw` never actually returns anything, since it jumps to a different place in the program. Thus, the type  $\beta$  here only serves to tell the type checker that ‘whatever type you want is fine’.

As a first example of `callcc`, consider the following program:

```
1 + callcc (fn c => 2 + 0)
```

This program computes the value 3. We are not making any use of continuations yet: while `callcc` here creates a continuation that we bind to variable  $c : \text{int cont}$ , we simply ignore this continuation and compute  $2 + 0$ . `callcc(fn c => 2 + 0)` then takes the result of this computation (2) and returns it unchanged.

Let us now try a more interesting example that actually makes use of  $c$ :

```
1 + callcc (fn c => 2 + (throw c 0))
```

This program computes the value 1. Here, we use `callcc` again to create continuation  $c$ .  $c$  represents the **orange** part of the code: ‘add 1 to some **int** value returned from `callcc`’.

Within the body of the continuation, we compute the expression ‘ $2 + (\text{throw } c \ 0)$ ’, but the **blue** part of the code never gets executed: as soon as execution runs into `throw c 0`, it takes the value 0 and plugs it into the continuation that we have bound to  $c$ , i.e., the **orange** part of the code. Thus, we only compute  $1 + 0$ , explaining the result of 1.

These kinds of semantics are tricky to represent in natural semantics rules, since they are intrinsically non-local, but researchers first came up with them in an attempt to give a formal semantics to language constructs like `goto` that can jump back and forth within code.

## 1 Letting Continuations Escape

In the above example, we are using the continuation like an exception, with the `callcc` operation acting like an exception **handler**.

However, continuations are more general than exceptions — they can jump backward and forward through the program. To see that, we have to let the exceptions escape from within the `callcc` operation. This is not entirely straightforward — for instance, we can’t write

```
val cont = callcc (fn c => c) (* Error *)
```

because the type system can’t infer a type for `cont`. Recall that `callcc` :  $(\alpha \text{ cont} \rightarrow \alpha) \rightarrow \alpha$ . In the above code, we pass an identity function to `callcc`, which means that we are asking the type inference system to find a type  $\alpha = \alpha \text{ cont}$ , but such a type does not exist<sup>2</sup>, so SML will give a type error for the above.

In other words, if we want to return a continuation from `callcc`, we have to pack it into another type that hides the  $\alpha \text{ cont}$  type. In SML, we can accomplish this only through an auxiliary **datatype**.

For example, consider the code in Figure ?? . Here, we package the continuation  $c$  into the datatype `clistmaker`, along with the number 100 and the empty list. As before, `callcc` initially just returns the packaged result `C(100, [], c)`. In the **case** deconstruction on line 5, we will thus match the branch on line 7. This branch then throws the

<sup>2</sup>In most languages with type inference; Ocaml actually allows such ‘cyclic types’.

```

1 open SMLofNJ.Cont;
2
3 datatype clistmaker = C of int * int list * clistmaker cont;
4
5 case callcc (fn c => C(100, [], c)) of
6   C(0, list, _) => list
7   | C(i, list, c) => throw c (C(i - 1, i :: list, c));

```

Figure 1: Building lists with continuations.

exception — effectively jumping back to line 5 with an updated value for the `clistmaker`, namely `C(99, [100], c)`. This process repeats until we have built a list containing all numbers from 1 to 100, at which point we return the list (line 6). The continuation `c` here consists of the code in orange:

```

case callcc (fn c => C(100, [], c)) of
  C(0, list, _) => list
  | C(i, list, c) => throw c (C(i - 1, i :: list, c));

```

with the entry point into the continuation again being the call to `callcc`.

## 2 Extended Example

We have now seen how to use continuations to skip execution and how to use them to implement loops. However, continuations also allow us to travel up and down the call stack. To see this, we look at a more involved example — evaluating a program in a simple language:

```

datatype value = STRING of string
               | INT     of int

datatype program = PLUS of program * program
                 | VALUE of value

```

Here, the intuition is that `PLUS` will add `INT`s and concatenate `STRING`s, but we want to treat it as an error if `PLUS` tries to combine a `INT` and a `STRING`. We can describe an evaluation function (without continuations) as follows:

```

fun eval (VALUE v)      = v
  | eval (PLUS(lhs, rhs)) = (case (e lhs, e rhs) of
    (STRING s1, STRING s2) => STRING (s1 ^ s2)
  | (INT i1, INT i2)      => INT (i1 + i2)
  | _                     => raise Exception
  )

```

Here, any type error will raise an exception.

We can rewrite this error handling with continuations. In Figure ??, we implement error handling again, but this time we add automatic recovery: if we run into an error while evaluating, we jump out of the evaluation to error handling code (line 20). This error handling code then provides a default replacement value and *jumps back into the middle of the computation that we just aborted*.

In more detail, the `eval` function now takes an extra parameter, an error handler of type `error`. When we find an attempt to add a string and an integer, we call `throw error_handler` (line 10) and also pass along a continuation to the current state of the evaluation, `rc`.

In `try_eval` we generate this `error_handler` via `callcc` in line 18. In the default (`INIT`) case, we extract the continuation to line 18 into `handler` and pass it to `eval` in line 21.

```

1 datatype error = ERROR of value cont (* encountered an error *)
2     | INIT of error cont
3
4 fun eval (error_handler : error) (expr) =
5     let fun e(VALUE v) = v
6         | e(PLUS(lhs, rhs)) = (case (e lhs, e rhs) of
7             (STRING s1, STRING s2) => STRING (s1 ^ s2)
8             | (INT i1, INT i2)      => INT (i1 + i2)
9             | _                     =>
10                 let fun recover rc = throw error_handler (ERROR rc)
11                     in callcc (recover) (* callcc-recover *)
12                     end
13             )
14     in e expr
15     end
16
17 fun try_eval (expr, default_value) =
18     let val handler = case callcc (fn x => INIT (x)) of (* callcc-handle *)
19         INIT (handler) => handler
20         | ERROR (recover) => throw (recover) (default_value)
21     in eval (handler) (expr)
22     end

```

Figure 2: Error handling with automatic recovery.

If `eval` encounters an error and throws the error handler at line 10, we thus jump back to line 18, this time taking the `ERROR` branch (line 20). At this point, we know that an error occurred and can handle this error however we see fit. In the code above, we throw the `recover` continuation, effectively jumping back to line 11 while passing along a default value. Since continuations preserve the full execution context (i.e., the call stack with all activation record instances), we can then continue executing.

For a more concrete example, let us call `try_eval` to evaluate `(3 + (4 + "foo"))`, which should give us an error during the addition `(4 + "foo")`:

```
val z = try_eval(PLUS(INT 3, PLUS(INT 4, STRING "foo")), INT 0)
```

We also specify a default value of 0. During execution, `try_eval` now calls `eval` with the continuation `error_handler` and the expression to evaluate. `eval` calls `e`, which calls itself recursively several times. Figure ?? shows the calls and returns.

When we evaluate `e (PLUS(INT 4, STRING "foo"))`, `e` recursively evaluates `e (INT 4)` and `e (STRING "foo")`, successfully. However, the return values have mismatching types, so we enter the error handling code in line 10. This code throws `error_handler` while passing along the continuation `rc`. `error_handler`, which is the continuation following `callcc` in line 18, now takes over to perform error recovery: it jump back to `recover` with the user-specified default value `INT 0`. This default value arrives in line 11, in place of the call to `callcc`, and from there is returned to the caller (i.e., it becomes the return value of `e (PLUS(INT 4, STRING "foo"))`). We then conclude evaluation as usual.

Our `try_eval` function implements just one possible error handling strategy that isn't particularly smart. For instance, if our default value were `STRING "hello"`, then plugging this default value into our example above would trigger *another* type error. Since continuations preserve the entirety of the execution context, we could write an error handler that tries out different values until it finds one that doesn't cause any errors further down, or one that minimises the number of errors.

Below is yet another (and much simpler) alternative approach, in function `eval_or_fail` that immediately returns `NONE` on failure, or otherwise lists a successful evaluation result into `SOME result`, exploiting the option

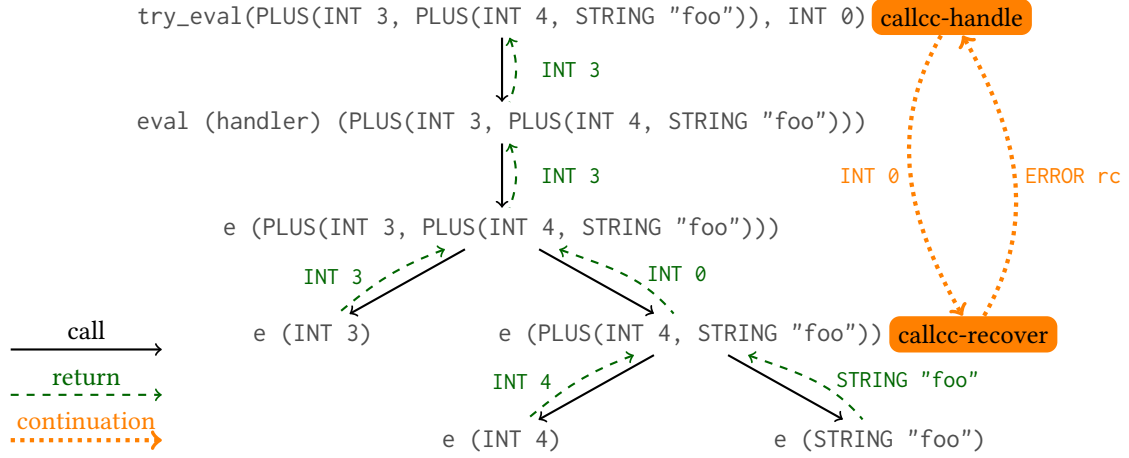


Figure 3: Calls and returns for our error handling example.

type constructor in SML.

```
fun eval_or_fail (expr) : value option =
  case callcc (INIT) of
    INIT (c) => SOME (eval (c) (expr))
  | ERROR (_) => NONE
```