

Difference between C's setjmp/longjmp and SML continuations

Setjmp saves the stack context (stack pointer and program counter, and some other registers, and optionally the signal masks) in a jump buffer. Longjmp then restores those registers, implicitly unwinding the stack, and jumps to the saved program counter.

Callcc saves the complete continuation (complete stack content, program counters, and some other registers). Throw then restores those registers, and the stack, and jumps to the saved program counter.

The primary limitation of setjmp/longjmp is that the jump buffer must *not escape* the function in which it was captured (the callee of setjmp). This is because the stack itself is not saved, only the stack pointer is. So if the jump buffer escapes, the required activation records might already have been overwritten or reclaimed. This usually leads to a segmentation fault.

Continuations have a potential drawback, though, because the activation records have to be "saved" when callcc is called. The potential memory and performance penalties can be minimized, though (as we have seen in one of our readings).

Control flow between callcc and setjmp *differs slightly*. Callcc will take a function *f* and return *f*'s return value, or throw's argument. Setjmp, though, takes no function. It returns 0, or longjmp's argument. One often uses setjmp's return value in the condition of an if statement, and calls *f* in the 0 branch.

Note: Another perceived aspect is that the values of local variables are not restored with longjmp. If one modifies a local variable after calling setjmp, this modification will be visible after a longjmp. In SML, which is a functional language, and thus rarely uses assignments, values are rarely modified anyways.

SML continuation example that is hard to do with C's setjmp/longjmp

The co-routine example from the lecture slides would be pretty tough to implement with setjmp/longjmp:

```

fun thread () = work () ; yield () ; thread ()
fun yield () = if random ()
                then ()
                else callcc (fn k => (enqueue k; dispatch ()))
fun dispatch () = let val head::rest = !queue
                  in queue := rest; throw head ()
                  end

```

Furthermore, any non-trivial example where a continuation escapes will be hard to emulate. The following example is trivial, one could convert it into a tail-recursive function without using continuations at all. But it shows the gist, that a continuation can escape the function it was created in. A straightforward translation of this example to setjmp/longjmp will quite certainly lead to core dumps.

```

fun main () = throw capture () ()
fun capture () = callcc (fn k => k)

```

Why are setjmp/longjmp not true library functions

When you call a well-behaved C library function it usually does some work and then returns. The language dictates that control flow at function invocation returns to the place of invocation. Function calls and returns are paired. Whenever you leave a function, all the things that happen at function entry are undone (activation record is deallocated, in C++ local variables are destructed). C's longjmp though, does not act like a function. It acts more like the C++ throw statement. It changes control flow. And it does that in a pretty unstructured way (goto-like). And both, setjmp and longjmp use internal knowledge of the generated code to implement their functionality (access/modify the internal state of the runtime system by reading/writing various registers).