

State Management in C Programs

Bill McCloskey

December 17, 2002

1 Introduction

Many programs can naturally be structured as state machines. Envisioning a program as a state machine can clarify its structure. Errors checking often becomes easier because the programmer must explicitly understand state transitions, and ensure that all possible error conditions are checked. Many systems would benefit from the use explicit state machines during the design phase.

Unfortunately, the job of encoding a state machine in C is tedious and error-prone. Such code often involves extensive use of callbacks or large *switch* statements. The current state must be saved in a globally accessible location. Saving and restoring this state can be tiresome.

A new tool, named *callcC*, has been written to address these problems. This paper will explore the design and implementation of *callcC*, with examples of its usefulness. The tool has been written in Objective Caml using the CIL infrastructure. It currently can perform state transformations on C programs, although the performance penalty of the tool is occasionally severe. I believe that, with optimizations, the tool will be capable of replacing many explicit state machines in real environments.

2 Motivation

The *callcC* tool is useful in the following settings.

Threading Threading is the most common form of concurrency available to programmers. Preemptively scheduled threads cannot be emulated using the *callcC* tool, since they require operating system support. However, cooperatively

scheduled threads can be implemented easily. In fact, the design of the *callcC* language favors a “threaded interpretation,” as will be explained below.

The inability to schedule preemptively is not entirely negative. Preemptively scheduled tasks require complex synchronization that is often unnecessary for cooperative threads. In addition, cooperative, user-space threads incur a smaller context-switching penalty.

Event-driven programming Event-driven programming is another form of concurrency. For larger systems, an event-driven model scales better than threading. Hybrid approaches are often used to take advantage of multiple processors, but these most closely resemble the pure event-driven model.

Unfortunately, it can be difficult to program an event-based system. Event handlers are forbidden to block, so that a linear block of code must be split into pieces. Any state information which crosses the boundary lines, where the code is split, must be saved and later restored. Breaking up code makes it more difficult to understand. In particular, it is much more difficult to infer what code will be executed, since event handlers can be registered almost anywhere. The *callcC* tool alleviates these problems by automatically breaking up the code.

Iterators When traversing a data structure, an iterator must save its state before each element is returned, and restore it when the next element is requested. *CallcC* supports suspension and resumption of functions, so that iterators can be

programmed naturally.

Lexers and Parsers A parser's state is stored on a stack. Recursive-descent parsers, which are simplest to write, use the program stack itself. Unfortunately, the use of an implicit stack prevents a recursive-descent parser from suspending itself in the middle of parsing. Suspension might be desirable when a parser needs more tokens or when it would like to pass parts of the parse tree to a different module.

Of course, the parser could easily call the tokenizer as a function, making `callcC` unnecessary. In that case, however, the tokenizer must save its own state explicitly. For example, *flex* generates lexers which store state in global variables. Although the state is saved explicitly, the lexer is no longer reentrant, a definite disadvantage. The lexer, parser, and tree processor are best understood as separate coroutines. The `callcC` tool allows them to be coded in C this way.

3 The Source Language

The `callcC` tool translates state management instructions into C statements. In order to make the tool as general as possible, the state management instructions are extremely basic. Program state can be divided into two categories.

- Some program state saves information about what code should be executed next. When state is represented explicitly, programs often contain an integer state variable. A `switch` statement can be used to implement a finite state machine this way. Other times, jump tables or function pointers are used. This kind of state will be called *program counter state*.
- Most program state is *data state*. Any extra data which must be saved across a blocking call falls into this category.

The `callcC` tool contains three instructions for state management. There are several interpretations of these instructions.

3.1 Coroutines and Threading

The first interpretation is in terms of threads or their cousins coroutines. A thread or coroutine is created with the `spawn(f, a1, a2, ..., an)` instruction. The thread begins by executing `f(a1, a2, ..., an)`. When execution of `f` completes, all memory used by the thread is automatically freed. Since these threads are cooperative, the `spawn` instruction begins executing `f` immediately.

Executing `callcc(f, a1, a2, ..., an)` suspends a thread or coroutine. There are a few complications, however. In order to reference a thread in the future, it must be identified. Before suspending a thread, `callcc` executes `f(k, a1, a2, ..., an)`. The `k` argument is a thread identifier. It must be saved by `f` so that the thread can be resumed later. Thread resumption is accomplished by `execute(k)`, which begins executing the thread right after the `callcc` statement.

This is clearly a very simple threading implementation. There is no way to wait for a thread to finish. There are no locking or communications primitives. All of these features could be implemented on top of `callcC`, though. Another restriction is that all threads must finish executing before the program finishes, or there will be memory leaks. Calling `execute(k)` when the thread `k` has finished is an error. Currently, it is also an error for a thread never to suspend itself.

An example coroutine to iterate over the nodes of a tree is given below. After each node is encountered, the coroutine suspends itself so that the caller can process the node. A real implementation would wrap the `callcc` and `execute` instructions to make them friendlier. Without `callcC`, a tree iterator would require an explicit stack. The code would be considerably more complicated.

```
struct iterator {
    int result;
    int done;
    void *k;
};
void set_cont(void *k, void **dest) {
    *dest = k;
}
```

```

void tree_iterator(tree_t tree,
                  struct iterator *iter)
{
    if (tree->left)
        tree_iterator(tree->left, iter);

    iter->done = 0;
    iter->result = tree->n;
    callcc(set_cont, &iter->k);

    if (tree->right)
        tree_iterator(tree->right, iter);

    iter->done = 1;
}

```

The iterator could be used as follows.

```

struct iterator iter = { 0, 0, NULL };

spawn(tree_iterator, tree, &iter);
while (!iter->done) {
    // ... use iter->result ...
    execute(iter->k);
}

```

3.2 Continuations

Another view of the callcC tool is that it adds continuations to standard C. A continuation represents “the rest of the computation.” The continuation itself saves the program counter state, and the environment attached to the continuation encapsulates the data state (as described above). Therefore, the k variable in the previous section can be considered a continuation. The *execute* instruction is similar *throw* in ML.

There are several differences between callcC and standard implementations of *callcc*. In Scheme or ML, *callcc(f, ...)* executes f and then returns to the caller. The callcC implementation returns to the most recent *spawn* or *execute*. Although it’s possible to emulate the latter using the former, the callcC method makes it much easier to implement coroutines. When a coroutine needs to suspend itself and return, it can simply use *callcc*.

One real restriction in callcC is that *execute* must be called on a continuation exactly once. Such continuations are called one-shot continuations [1]. To call it zero times would cause memory leaks. To call it more than once would possibly lead to memory errors. This restriction means that backtracking and nondeterminism cannot be implemented with continuations. Exceptions, which may require a continuation to be thrown away, also cannot be implemented directly.

Changing callcC to support exceptions would be fairly easy. An instruction, *kill(k)*, to free a continuation would be sufficient. Adding backtracking would be far more difficult, since it requires that a stack be copied. Copying the stack is impossible in C, since stack variables may be referenced by outside pointers.

3.3 State Machines

CallcC can also be viewed in terms of state machines. A call to *spawn(f, a₁, a₂, ..., a_n)* creates a state machine. Any computations made by f before suspension are simply computing the start state of the state machine. After the machine has been suspended, it is fixed in a state k and the rest of the program continues to execute normally.

When the rest of the program wishes the state machine to make a transition, it calls *execute(k)*, which computes, based on k and the global state of the program, a new state k' . Since *execute* takes no parameters, the global state of the program is used to determine the proper transition to make.

Although the state machine perspective is not very useful operationally, I believe that it might be helpful for program analysis.

4 Transformation

It is easiest to understand the transformation itself using the threading interpretation. The *spawn* instruction must create a stack for the new thread. The *callcc* statement places the current stack in the k variable and then switches to another thread. *execute* also switches to another thread.

All information about the thread is saved on its stack. This includes:

- Function arguments and local variables
- A location to save a function's return value
- A return pointer, identifying the code which should be executed when a function returns
- A resume pointer, identifying the code which should be executed when a thread is resumed

Parts of the transformation resemble a CPS translation. However, only certain functions take part in the translation. A function is said to be *preemptable* if it uses *callcc* or if it calls a preemptable function. The preemptable property can easily be discovered by examining the call graph. Only preemptable functions are CPS translated.

The resemblance to a CPS translation is as follows. When a preemptable function is called, it is not expected to return until the thread is suspended. Thus, a *return* statement is inserted after every call site to drop control back to the most recent *spawn* or *execute*. The current continuation (meaning, in this case, the code to execute after the function call, without the environment) is passed on the thread stack to the called function. When it is finished, the called function will run this continuation.

When a thread is suspended with *callcc*, the current continuation is pushed on to the thread stack. Then a *return* statement returns control to the most recent *spawn* or *execute*, since any preemptable functions simply return immediately. Implementing *execute* is simply a matter of popping a continuation off the stack and running it.

The *spawn* call must allocate space for the stack. In the current implementation, it allocates a fixed-size block. If the stack becomes too large, the program will crash. I will address this issue below. In addition, *spawn* pushes a continuation for the stack destructor on the new stack. This ensures that the last operation of any thread will be to free its own memory.

Any preemptable function must access its arguments and local variables via the thread stack. The

transformation generates, for each preemptable function, structure declarations for local variables and formal arguments. Space for these structures is reserved when a function is called. Variable accesses are modified so that they are made through these structures.

In several cases, it is necessary to push a continuation on the stack. A continuation must identify not only the function to execute next, but where in the function to begin executing. Functions are identified by function pointers, and the rest is done using an integer. Each possible suspension point in a function, including any place where a preemptable function is called, is given an index. These indices are passed around as part of the continuation. When a preemptable function is called, it is passed the index. It uses *goto* statements to transfer control to the proper location.

All operations on the thread stack are performed using an external library. An example program is given below. Figure 1 shows the translation of the preemptable function *thread*, and Figure 2 shows the transformation of *main*.

```
void *global_k;
void set_cont(void *k) {
    global_k = k;
}
int thread(int x) {
    int y = 7;
    printf("in thread\n");
    callcc(set_cont);
    printf("finishing thread:"
           "x = %d, y = %d\n", x, y);
    return 5;
}
int main(int argc, char *argv[]) {
    spawn(thread, 10);
    printf("in main\n");
    execute(global_k);
    return 0;
}
```

The output of this program is:

```
in thread
in main
finishing thread: x = 10, y = 7
```

5 Related Work

The first account of coroutines appears to be [2]. Conway focused on the separation of tasks accomplished by using coroutines. He used them to implement a COBOL compiler in assembly language. He was interested in the fact that, by simply changing the way the coroutines communicated, the compiler could be made either one-pass or two-pass. Since the compiler was written in assembly, there was no need for any sort of stack.

The string processing language SL5 has a coroutine mechanism quite similar to `callc` [5]. SL5 has a number of interesting eccentricities, but the most pertinent one is its procedure call style. SL5 separates procedure activation into three operations: environment creation, argument binding, and resumption of execution. Environment creation allocates storage for a coroutine. Argument binding “transmits” a set of parameters to the coroutine. Arguments can be retransmitted even after the procedure has begun executing. Procedures can arbitrarily be suspended or resumed. The language provides a fairly intuitive semantics for transferring control between coroutines. Except for the unusual argument passing model and the need for multi-shot continuations, an SL5-like procedure call style could easily be implemented with `callc`.

Although there is a great deal of research on continuations, the concept of subcontinuations most closely resembles `callc` continuations [6]. A language with subcontinuations contains a *spawn* operation, similar to the one in `callc`. However, every subcomputation created by *spawn* is passed a *controller*. Invoking the controller is similar to invoking *callcc* in `callc`. However, each subcomputation receives its own controller. Thus, when viewing the subcomputations as a hierarchy according to the way they were spawned, it is possible to suspend any ancestor computation—not just the parent. It is not immediately clear to me how useful this additional functionality would be.

In [1], the concept of a one-shot continuation is presented. Interestingly, the authors of [7] show how one-shot subcontinuations can be implemented using threads. They assume the presence of preemptively scheduled threads with standard locking primitives. The `callc` tool is somewhat similar, except that it

does not allow hierarchical subcontinuations (as described above) and it uses cooperative threads.

A model for event-driven programming using continuations is presented in [8]. A special call to wait for events blocks the computation until an event arrives, and then dispatches the event using a table of continuations. Interestingly, the author also allows for a restricted form of preemptive concurrency by allowing a thread to give its CPU time to another thread until an event arrives, at which point the thread is resumed.

Regarding performance numbers, [3] gives some promising results. The authors added continuations as a control transfer mechanism to the Mach 3.0 system. Rather than saving registers and the stack when it blocked, a kernel thread would specify a continuation. Consisting only of a function to execute, continuations were much lighter weight. By manually changing some blocking points to continuations, the authors decreased space usage and increased performance.

Finally, it is interesting to compare `callc` to a user-space threading library, since `callc` can emulate threads. Although the current implementation of `callc` is likely to be significantly slower than a thread package, it does have portability benefits. The Pth library [4] is intended to provide portable user-space threading across a variety of Unix platforms. Unfortunately, the final result is a mishmash of hacks to overcome the shortcomings of *setjmp* and *longjmp* and to deal with signals properly. `Callc` is interesting because it is perhaps the only *completely* portable threading library in existence.

6 Future Work

There are many areas where `callc` could be improved. Most importantly, the tool produces inefficient code. I have not had the time to benchmark it sufficiently, but some preliminary work indicates that execution time can triple. Benchmarking is difficult because two copies of the benchmark must be written, one using `callc` and the other using explicit state.

Additionally, I would like to explore alternate ways

of saving the stack. The current scheme of allocating a fixed-size block of memory is clearly deficient. Finally, I have several ideas about program analyses and other techniques involving callC.

6.1 Optimizations

One obvious problem with callC is that it saves too many variables on the stack. Even temporaries generated by CIL are accessed from the stack. Only variables which are live across a suspension point need to be saved.

Further, not all variables which are saved on the stack must be accessed from it. Accessing variables directly from the stack circumvents the register allocator. It would be better to load most variables from the stack into temporaries when a coroutine resumes, and save them before suspending. Only those variables which do not have their address taken and which are not arrays can be accessed this way, though.

A third obvious optimization is to remove the safety checks from the stack library, and perhaps to inline the stack operations. The current implementation assigns a type to every item on the stack and enforces type safety with assertions.

Finally, it might be useful to save program counter state in one variable instead of two. A coroutine would be broken into multiple functions, with each one executing all the code after a certain suspension point. After the change, the *switch* statement at the top of each preemptable function could be eliminated. Unfortunately, creating a new function for each suspension point could cause a quadratic increase in the size of the program, which would harm cache locality. A heuristic to balance these two considerations might be helpful.

6.2 Stack Management

There are a number of alternate stack allocation mechanisms possible. The simplest is to allocate a new block of memory on the heap for each stack frame. A more efficient strategy would begin by allocating small blocks to hold only a few stack frames, and increase the block size geometrically. This has

the advantage that simple tasks like iterators take little space, while recursive procedures will never overflow the stack or waste time in memory allocation. Unfortunately, pushing and popping on such a stack have a larger overhead, since the blocks would have to be tracked in a list.

Ideally, when function pointers and recursion are not involved, it is possible to conservatively determine the size of the stack frames needed by a function and all its descendants in the call graph. In this case, allocation becomes easy. One could imagine that in some rare cases, the average space used by a thread would be much less than the maximum size. A procedure could determine both the minimum and maximum stack space needed and heuristically decide whether to make the optimization.

It would make sense to optionally implement the stack non-portably. By directly manipulating the stack and frame pointer registers, callC could approach or exceed the performance of any user-space threading library. Since it could estimate the stack usage of a thread, it would have the potential to be much lighter weight as well.

Finally, I would like to implement an automatic transformation to make an entire module reentrant. The Microsoft Visual C++ compiler apparently allows global variables to be declared as thread-local. Consider, for example, a random number generator which stores its seed in a global variable. The transformation would keep thread-local storage in the continuation k , along with the stack. The seed variable would be stored there, and all references to it would be changed to access the thread-local area. Such a transformation would be extremely useful for existing code.

6.3 Types

The current implementation of callC forbids the use of function pointers to preemptable functions. In the presence of such pointers, it would be impossible to determine whether a function call was calling a preemptable function or not. I propose to fix this problem by forcing the programmer to include an attribute in the type of every preemptable function.

Since it would be tedious to annotate all preemptable function types, a type inference algorithm would be useful. Unfortunately, since programmers often cast function pointers, some annotation would still be necessary. Also, it could be dangerous to require annotations only in rare cases. I would like to investigate this issue further.

6.4 Analysis

There are a number of ways to make the tool even more useful for programmers. One problem with cooperative multitasking is that tasks may hog the CPU. A profiler could be written to watch the event loop and determine which thread was yielding too infrequently. By observing the most frequently executed parts of a thread, the profiler could even suggest places to insert suspension points.

Another possible analysis could find locations in a program where errors were not checked. In the state-machine interpretation of the `callc` tool, there are multiple transitions out of any state. Normally, the transition type is stored in a globally accessible location before a call to `execute`. To increase safety, a number of possible transition types could be defined, and the correct one would be passed to `execute`. After resuming from a `callc`, the thread would be required to check all transition possibilities, much as a Java program is required to catch all exceptions. Interestingly, after these annotations were in place, a program understanding tool could infer the structure of the program's implicit state machine.

6.5 Systems

I also would like to explore several avenues which pertain only to large systems. One interesting idea is to manage ownership of an object using a thread. With extremely lightweight threads, it becomes feasible to create one thread for every object of a certain type in the system. Such a technique might simplify memory management for long-lived objects and make garbage collection unnecessary.

A disadvantage of user-space threads is that they cannot take advantage of multiple processors. By creating one kernel thread for each processor, and run-

ning many user-space threads in each kernel thread, the problem would be solved. Unfortunately, this brings up a number of difficult synchronization issues. I would like to investigate this area, especially in the context of the SEDA framework for servers [9].

7 Conclusion

The benefits of adding `callc` to C have been discussed above. Coroutines, event-driven programs, and iterators become much easier to code. Since the implementation is so inefficient, though, one could ask why a standard threading library would not be more desirable. Using the technique in [7], threads can be used to implement the same kind of continuations as those in `callc`.

One unique benefit of `callc` is its portability. Not only is it architecture independent, it makes no assumptions about signal handling and it does not require thread-safe libraries. Many GUI toolkits are not reentrant and cannot be used with threads; `callc` has no such issues. `Callc` works on embedded systems, and with the right stack management strategy, it would use far less memory than threading.

In the other direction, by sacrificing portability, I think the efficiency of `callc` could increase dramatically. A `spawn` call would reduce to `malloc` and context-switches would be a matter of swapping registers. With the ability to approximate the stack size, `callc` could be the most light-weight threading library available.

The current `callc` implementation has been tested somewhat thoroughly. I wrote a simple networking server with it to parse XML commands and send responses. I also successfully ran the tool on a `flex`-generated lexer, so that the lexer would suspend execution while awaiting input characters. Simple iterators are known to work too. Ultimately, although the tool offers few new research ideas, I think it could make many C programming tasks much easier.

References

- [1] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Conference on Programming Language Design and Implementation*, June 1996.
- [2] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7), July 1963.
- [3] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136. Association for Computing Machinery SIGOPS, 1991.
- [4] Ralf S. Engelschall. Portable multithreading. In *USENIX Annual Technical Conference*, 2000.
- [5] David R. Hanson and Ralph E. Griswold. The SL5 procedure mechanism. *Communications of the ACM*, 21(5), 1978.
- [6] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson. Subcontinuations. In *LISP and Symbolic Computation 7*, volume 1, January 1994.
- [7] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. Threads yield continuations. In *LISP and Symbolic Computation 10*, volume 2, May 1998.
- [8] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, January 1997.
- [9] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, 2001.

```

struct thread_info { int y ; };
struct thread_args_info {
    int *result ;
    int arg0 ;
};
void thread(continuation_t k , int jump )
{ struct thread_info *v ; // local variables
  struct thread_args_info *a ; // arguments
  void (*tmp7)(continuation_t k , int jump ) ;
  int tmp8 ;

  if (jump == 0) {
    // allocate space for local variables
    cont_add_data(k, 4, 10);
  }
  // get pointers to locals and args from stack
  // 4 == sizeof(thread_info), 8 == sizeof(thread_args_info)
  v = (struct thread_info *)cont_top_data(k, 4, 10);
  a = (struct thread_args_info *)cont_second_data(k, 8, 4, 5);
  switch (jump) {
  case 1:
    goto Cont1;
  }
  v->y = 7;
  // suspend
  cont_push_int(k, 1);
  cont_push_function(k, & thread);
  set_cont(k);
  return;

Cont1: /* CIL Label */
  printf("finishing: x = %d, y = %d\n", a->arg0, v->y);

  if (a->result != 0) {
    *(a->result) = 5;
  }
  cont_pop_data(k, 4, 10);
  cont_pop_data(k, 8, 5);
  tmp7 = (void (*)(continuation_t k , int jump ))cont_pop_function(k);
  tmp8 = (int )cont_pop_int(k);
  ((*tmp7))(k, tmp8);
  return;
}

```

Figure 1: The transformed *thread* function

```

int main(int argc , char **argv )
{ continuation_t tmp3 ;
  struct thread_args_info tmp4 ;
  void (*tmp5)(continuation_t k , int jump ) ;
  int tmp6 ;

  // spawn(thread, 10);
  tmp3 = (continuation_t )cont_allocate();
  tmp4.arg0 = 10;
  tmp4.result = 0;
  cont_push_int(tmp3, 0);
  cont_push_function(tmp3, & cont_destructor); // cont_destructor frees memory
  cont_push_data(tmp3, & tmp4, 8, 5);
  thread(tmp3, 0);

  printf("in main\n");

  // execute(global_k);
  tmp5 = (void (*)(continuation_t k , int jump ))cont_pop_function(global_k);
  tmp6 = (int )cont_pop_int(global_k);
  ((*tmp5))(global_k, tmp6);

  return (0);
}

```

Figure 2: Translation of the *main* function