Exchange of Control

John Skaller

April 8, 2020

1 Subroutines

All modern programming languages make use of an artefact known as the *subroutine*. A subroutine is designed to form a reusable component of code. It is a sequence of instructions that a computer can execute to accomplish a particular task. By being both reusable and modular, it simplifies the practice of building and maintaing programs.

A subroutine consists of a few important details: it has a fixed starting address known as the *entry point*; it takes a value parameter, which is provided (or passed) by the caller; and it uses the processor's *machine stack* as temporary storage (this is where it holds parameters and short term variables).

When a processor executes a subroutine, it starts with a `call` instruction. This instruction does two things simultaneously: It moves the instruction pointer to the subroutine entry point (akin to a `jump`), and it saves a pointer to the *return address* (or calling location)—this is like a trail of breadcrumbs the CPU can use to find its way back out of a maze of procedure calls. When we talk about *control*, or *control flow*, we mean this trail of crumbs.

Once inside the subroutine, the processor advances the stack pointer and allocates space for local variables on the machine stack. The subroutine itself may perform calculations or even call other subroutines to do its work.

When the subroutine is done executing, it 1) cleans up temporary data (parameters, local variables, and extinct return addresses), 2) saves a return value to the machine stack, and 3) passes control (jumps) to the return address supplied by the caller.

What we've described, so far, is a stack-based machine. It's a bit like an onion, with interleaving nested layers of data and control information, growing outward. Calling a subroutine adds a layer, and returning from a subroutine peels off a layer.

The important point, insofar as this paper is concerned, is that the control and data information are tightly integrated, coupling *control flow* with *data*

flow. This coupling demands a caller/callee (or master/slave, client/server, etc.) relationship which, as we shall see, may not be necessary or beneficial.

What makes this machinery popular is that it is affected by precisely two machine registers: the *program counter* and the *stack pointer*. It is also highly efficient, allowing immediate storage for parameters, subroutine locals, return addresses, and return values.

Furthermore the mechanism supports writing *re-entrant* code. That is, subroutines which may be recursive and reused simultaneously by many threads of control. Each thread allocates its own stack and can execute independent of the other threads. In most programming systems, this is how you get concurrency. Threads operate independently, with their own local stacks, and they communicate via global data on the *heap*.

Despite being successful, this mechanism does have some downsides. The almost universal adoption, to exclusion of others models, has twisted modern programming into an extremely ugly shape.

The serious deficiencies of the machinery often exhibit with complex problems. Subroutines are repeatedly called but require little more than state calculations from prior calls. Message handling callbacks (e.g. in GUI systems) and functional programming with higher order functions, often exhibit evidence of this.

In this paper we will work through two examples which highlight the serious deficiencies of the subroutine call model. These examples are: 1) a study on a code refactoring technique known as *control inversion* and 2) a functional *fold* with an equivalent procedural iteration.

2 Control Inversion

Control inversion is a refactoring technique which is used to reverse a master/slave relationship. The slave becomes the master, and the master the slave. Whilst demonstrating this keep in mind that this reverses the direction of control flow exhibited by the call/return protocol.

```
proc slave1 () { println$ "slave1"; }
proc slave2 () { println$ "slave2"; }
proc slave3 () { println$ "slave3"; }
proc master () {
  slave1();
  slave2();
  slave3();
}
```

Let us begin with the simple situation illustrated above. A master procedure

calls three slaves, which are used to produce a fixed sequence of effects. We will show how to *control invert* the subroutine call to `slave2` so that it becomes the master and the master a slave:

```
proc slave1 () { println$ "slave1"; }
proc slave3 () { println$ "slave3"; }
proc newmaster () {
  slave1();
  println$ "slave2";
  slave3();
}
```

What we have done here is as follows:

1. `newmaster` must first perform all the tasks that `master` previously performed up to the point where `slave2` was called;
2. Then it does its own work; and
3. finally, all the work `master` would have done after `slave2` returned.

Let us study a slighly more difficult problem, where the utility of this procedure becomes a little more evident. The following code could be used to draw a series of line segments on a canvas. You will notice that there is a variable called `first`, along with `last_x` and `last_y`. Naturally, since a line consists of two points, we cannot hope to draw anything until the second point.

```
proc dispatcher() {
  while true do
    var x,y = get_mouse_pos();
    call draw(x,y);
  done
}

var first = true;
var last_x, last_y = 0,0;

proc draw(x:int, y:int) {
  if not first do
    draw_line (last_x, last_y, x, y);
  else
    first = false;
  done
  last_x,last_y = x,y;
}
```

The slave routine here (`draw`) illustrates very bad programming style: it saves

essential state (`first`, `last_x`, and `ast_y`) in global variables and hence is not re-entrant. But there is no alternative whilst the master dispatcher insists on calling it without an auxilliary state variable to hold the data.

Let us see what happens with control inversion:

```
proc draw() {
  var first = true;
  var last_x, last_y = 0,0;
  while true do
    var x,y = get_mouse_pos();
    if not first do
      draw_line (last_x, last_y, x, y);
    else
      first = false;
    done
    last_x,last_y = x,y;
  done
}
```

This is clearly a much nicer piece of code because the state variables are now local to the draw routine.

In a more general GUI setting many programmers may have noted the difficulty of maintaining complex state in callbacks. This is because the machine stack, which is normally used to host local variables containing state information, is lost each time the callback returns control. This in turn is due to the coupling between the flow of control and data, the advantages of which are substantially lost in a callback slave.

This also explains why the Simple Direct Media Layer (SDL) library is so cool. It does not use callbacks! Instead, client *master* code fetches events from the event queue directly. In other words, the library *control inverts* the normal dispatcher subroutine call.

Now let us go back to our example, and observe that we have cheated a little. What we have actually done is just inline the subroutine into the dispatcher, lifting the global variables onto the stack as locals. We haven't *really* control inverted it, have we?

We need a more difficult example to really show what is going on. We will pick the well known function `fold_left` from functional programming. Folds are one of the highlights of the power of functional programming, but we will show that they are essentially bad.

A list fold left looks like this:

```
fun fold_left
  (f:double -> int -> double) // processing function
  (init:double) // initial value
  (lst: list[int])
=
{
  var result = init;
  for elt in lst do
    result = f result elt;
  done
  return result;
}
```

This is a procedural implementation, but it is pure, having no side effects and depending only on its arguments. A recursive implementation is also possible, but we will see it presents extreme difficulties a bit later:

```
fun fold_left
  (f:double -> int -> double) // processing function
  (init:double) // initial value
  (lst: list[int])
=>
   match lst with
   | Empty => init
   | Cons (head, tail) =>
     fold_left f (f init elt) tail
   endmatch
;
```

It seems simple to use a `fold_left`, for example to find the sum of elements in the list:

```
var lst = (1,2,3,4).list;
var sum = fold_left
  (fun (init:double) (elt:int) => init + elt.double)
  0.0
  lst
;
```

however there is a very serious problem here: the processing routine is a slave. It is stateless. The init value does provide a state variable, however.

But consider now a weighted sum, where the first element has weight 1.0, and each subsequent element a weight of one more. Now our fold will not work

unless we modify the type of the init variable to include the weight, or we store the weight in a global variable.

In the first case, the fold then returns the wrong answer, as we have to strip off the unwanted weight from the final result.

```
var lst = (1,2,3,4).list;
def var wsum,_ = fold_left
  (fun init elt => init.0 + init.1 * elt.double)
  (0.0,1.0)
  lst
;
```

This may look easy, but the need to strip off the state variable clearly shows that, in general, fold is the wrong function to use. The correct function requires a second argument to extract the result from the state variable.

However that too is wrong, because the type of the state variable is now dependent on the implementation detail of the processing function. You can't win that way either. In functional programming, there is no possible solution. The type of the state variable is an implementation detail and should be known only to the processing object.

Note that merely saying you can strip off the state variable is also incorrect. If the fold is nested inside a heirarchy of functions its use is an implementation detail, and the use in the implementation doesn't strip off the state variable, well, you're screwed.

In fact the problem is solved easily with procedural code and a mutable variable global to the fold operation but local to the function calling the fold. Procedural programming is superior here precisely because no change in type is involved because the code is *not* pure. Of course one loses referential transparency and re-entrancy, but that could be better than the rigid purely functional straight jacket enforced no solution.

The problem of maintaining the state variable in a fold rapidly becomes overwhelming as the problem becomes more complex. It is even worse when the fold is over a tree, for example, especially if the tree has a different type of data attached to branches, nodes, and leaves, because then we will require three callbacks slaves which must coordinate their operation with a shared state variable.

Experienced functional programmers soon abandon either folds or purity! In a complete problem it is easier to write a single pure recursive routine specific to the problem, than it is to maintain the state variable, even though that is always possible.

Now, let us see what happens with control inversion! We will note that the inverted fold is intrinsically procedural, not functional. This is the kind of solution with which users of C++ STL will be familiar:

```
// list iterator
fun at_end (it: list[int]) =>
  match it with
  | Empty => true
  | _ => false
;
fun next (it:list[int]) {
  match it with
  | Empty => Empty[int]
  | Cons (_, tail) => tail
;
fun get (it: list[int]) =>
  match it with
  | Cons (head, _) => head
;

// sum
fun sum (lst: list[int]) {
  var result = 0.0;
  var it = lst;
  while not (at_end it) do
    result = result + it.get.double;
    it = it.next;
  done
  return result;
}
```

This is clearly a much better solution for the programmer of the sum operation. But for the programmer of the iterator there is a lot more work.

The reason is simple: the sum routine in the control inverted fold is a master. The iterator, on the other hand, is a slave.

Clearly control inversion is a useful technique, in such cases as the fold, when the processing function is complex and the data structure scanner is simple. Had the scanned data structure been a simple tree without parent links, the data structure scanner would be horribly complicated, and far better implemented by a recursive functional subroutine that calls the slave processing function as required.

However, this implementation is very bad if the slave is also complicated, and the control inverted procedural solution simply shifts the work load onto the scanning routine.

There must be a better way!

3 Coroutines

The solution to the problem above is so simple it is not funny! We must get rid of slavery altogether so that *both* the scanner and processing function are masters.

It may seem impossible to do this, since one must call the other: using control inversion we have shown we can *choose* which is the master but the other routine must be a slave.

Right?

The answer is yes, in a dumb programming model out of the 1970s utilising stacks and subroutine calls.

The pragmatic solutiuon in dumb languages is universal: use pre-emptive threads. This works, at the cost of introducing synchronisation issues. If you doubt that this is the *only* known solution with dumb languages, just examine real complex programs: my copy of Firefox, for example, is running over 100 threads. This is evil incarnate, because on a quad core intel processor with hyperthreading, there are 8 logical processors, and the correct number of threads to run is approximately nine. Anyone running more threads than that doesn't have any idea how to program correctly.

Who is at fault here? YOU are. If you're a slave to the purely functional concept, its you're fault. If you're a slave to the object oriented concept, its your fault. You have to learn new ideas and you're resisting!

Actually its not a new idea. The PDP-11 had the correct instruction, namely, branch and link. This instruction *swaps* control between two registers. The current continuation is saved in one register whilst another continuation is loaded from a second register.

But, this machinery is too primitive for modern programming. We must recover the most fundamental artefact of the modern idiom: local variables.

To do this we use coroutines. Coroutines are simultaneously masters and slaves, so it is more precise to say that they are peers since there is no heirarchy.

To make coroutines, we use another procedure known as control neutralisation. Control neutralisation removes subroutine calls and therefore master slave relationships. In real code, you can keep subroutines where they're useful, in both procedural and functional code: coroutines require some extra machinery which is not well supported on modern processors, and of course real programmers have to deal with legacy libraries.

In fact I contend all programs should consist of a balanced mix of functional code, procedural code, and coroutines. What is important is that the mix can be adjusted by two refactoring processes: control inversion and control neutralisation.

Here is a neutralised fold:

```
proc cofold_left
  (lst: list[int])
  (out: oschannel[opt[int]])
{
  for elt in lst do
    write (out, Some elt); done
  write (out, None[int]);
}
```

What, you say? That's just an iterator which writes the elements down a channel. It uses an option type so that the end can be indicated.

You got it! Simple eh?

Ok so what about the processing routine?

```
proc sum
  (inp: ischannel[opt[int]])
  (out:oschannel[double])
{
  var result = 0.0;
next:>
  var maybe_elt = read inp;
  match inp with
  | Some v =>
    result = result + v.double;
    goto next;

  | None =>
    write (out, result);

  endmatch;
}
```

Although this routine looks a bit complex it is intrinsically simple because it has the form of a master. That is, it maintains state in local variables.

Now, these two components can be coupled like this:

```
begin

  // test data
  var lst = (1,2,3,4).list;
```

```
    //construct communication channels
    var intinp, intout = mk_ioschannel_pair[int] ();
    var doubleinp, doubleout = mk_ioschannel_pair[double] ();

    // connect and spawn
    spawn_fthread { cofold_left lst intout; }
    spawn_fthread { sum intinp doubleout; }

    // get result
    var result = read doubleinp;
  end
```

Again, although the coupling machinery looks complex, this is because it is extremely powerful and we have not shown some of the syntactic sugar available in Felix to make it look simpler in special cases.

Critically, both the scanner and processing routine are coroutines so both can be written naturally. The functional programming fold has a good point that the processiung routine is independent of the data structure being scanned, or how it is sequenced, it will work in a right folrray fold as well (although the results may differ depending on whether the operation is associative).

The coroutine solution shares this property. Furthermore it is clear that whilst the coroutine solution presented here is imperative, the two procedures are entirely pure: there are no side effects, and the routines do not depend on anything but their arguments.

4 Automatic Control Neutralisation

The Felix compiler automatically neutralises procedure calls. It does this "behind the scenes" so that programmers are not aware of it. This is how Felix procedures can also act as coroutines and communicate with channels.

The machinery works like this: each procedure is divided into sections called resumptions. A resumption is, in effect, a delimited continuation. The local variables of the procedure are allocated as an object on the heap, instead of using the machine stack, however the object is called a stack frame anyhow, even though it isn't.

Now, when a procedure is called, the callers heaped stack frame pointer is saved in the heap object. That is the `this` pointer of the C++ object. When the compiler sees a return statement, it is replaced by a C++ return which returns the callers return pointer.

A call statement is effected by saving the current contuation in a special non-static variable of the stack frame called, appropriately, `pc`. This is normally

just an integer. Then the caller create a new object on the heap with C++ new operator, stores the context into that object, calls the objects `call` method to store the argument to the procedure into the variables of the stack frame representing the parameters, and save its own `this` pointer into the target object variable `caller`. It then simply returns! It does not call the procedure! Control returns to the scheduler.

Now, control is returned to the procedure, it is done by the scheduler calling the procedure's `resume` method. This method simply performs a C switch on the value in the `pc` variable so control jumps to where it last left off and the procedure can continue.

The scheduler maintains a list of active fibres, each fibre being an object containing the current continuation of a procedure, that is, its `this` pointer. Usually the scheduler does nothing more complicated than this:

```
    while (p) p = p->resume();
```

that is, it repeatedly resumes the current continuation of the currently active fibre. So you can see now, to call a routine we just make a new heap object and returns its pointer to the scheduler which then invokes its resume method, and to return from a procedure we just return the callers this pointer and the scheduler, blissfully unaware of what is going on, just keeps on running stuff.

You may wonder, why bother to do this? In fact in ordinary old code, the compiler optimises all that stuff away and just uses ordinary C functions. But if the code does I/O on schannels, something different happens!

The I/O request is stored in a special varible in the procedure stack frame which is a pointer to a service request. The scheduler checks, before resuming, if a service request has been made. If there is a service request it is performed.

For schannel I/O reads and writes cause the channel to be examined. If the request is a read and there is a pending write, the scheduler copies the data, which is always just a heap address, from the write to the reader, and pops the writer off the schannel wait list onto the active fthread list. If the request is a write and there is a waiting reader than the same happens, except that the current fibres is always the reader in Felix, and the write has to wait for the reader to consume the data.

If there is not a matching I/O request on the channel the requestor is pushed onto the channel and removed from the scheduler. The scheduler then picks another active fibre to run. If there isn't one, the program terminates.

Because of this machinery, procedures are coroutines which can *exhange control* as well as data. In effect the compiler control inverts all procedure calls by making the procedure into a slave: the slave function is the `resume()` method. The master function is of course the scheduler. However to the programmer, the

program has been mechanically control neutralised, not inverted: the callbacks are hidden in the run time library and not visible.

Fibre look like threads, except they're cooperative not pre-emptive. The importance of mastery is absolute. No programmer can write slaves of more than a medium level of complexity because of the huge burden of manually maintaining state due to the loss of persistence of local variables. This is not only as true in functional programming it is even more true.

Functional programming is evil. The enforced model of slavery destroys the paradigm. Procedural programming is actually easier, despite the loss of referential transparency, because models like OO teach programmers to localise state variables. But it is still hard.