Programming with Coroutines

John Skaller

May 27, 2017

Chapter 1

Introduction

Coroutines are not a new concept, however they have been ignored for far too long. They solve many programming problems in a natural way and any decent language today should provide a mix of coroutines and procedural and functional subroutines, as well as explicit continuation passing.

Alas, since no such system exists to my knowledge I have had to create one to experiment with: Felix will be used in this document simply because there isn't anything else!

1.1 What is a coroutine?

A coroutine is basically a procedure which can be spawned to begin a fibre of control which can be suspended and resumed under program control at specific points. Coroutines communicate with each other using synchronous channels to read and write data from and to other coroutines. Read and write operations are synchronisation points, which are points where a fibre may be suspended or resumed.

Although fibres look like threads, there is a vital distinction: multiple fibres make up a single thread, and within that only one fibre is ever executing. Fibration is a technique used to structure sequential programs, there is no concurrency involved.

1.2 A Simple Example

The best way to understand coroutines and fibration is to have a look at a simple example.

1.2.1 The Producer

First, we make a coroutine procedure which writes the integers from 0 up to but excluding 10 down a channel.

```
proc producer (out: %>int) () {
  for i in 0..<10
    perform write (out, i);
}
```

Notice that as well as passing the output channel argument `out`, there is an extra unit argument `()`. This procedure terminates after it has written 10 integers.

1.2.2 The Transducer

Next, we make a device which repeatedly reads an integer, squares it, and writes the result. It is an infinite loop, this coroutine never terminates of its own volition. This is typical of coroutines.

```
proc transducer (inp: %<int, out: %>int) () {
  while true do
    var x = read inp;
    var y = x * x;
    write (out, y);
  done
}
```

1.2.3 The Consumer

Now we need a coroutine to print the results:

```
proc consumer (inp: %<int) () {
  while true do
    var y = read inp;
    println y;
  done
}
```

1.2.4 Purity

The first two coroutines are *pure* because they depend only on their arguments, and interact with the outside world entirely through synchronous channels. The do not modify variables in their environment, and they do not depend on variables in their environment. The consumer, however, has a side effect, namely printing values to standard output.

Purity is an important property which provide modularity and encapsulation and allows one to reason about them locally. This is a vital information hiding property which is also possessed by pure functions, where it is known as functional abstraction. For coroutines, we would call this cofunctional abstraction, however there's a problem: functions are abstracted to function types. However the behaviour of a coroutines depend not just on the data type of the channels, but also on the order in which operations are performed on these channels, and that information should be approximated and symbolised by a *control type*. Alas, we do not have a suitable type system.

1.2.5 Synchronous Channel Construction

Now, let us see how we can use these coroutines in the obviously intended way! First we have to make some channels to connect the devices:

```
proc doit () {
  var ich1, och1 = mk_ioschannel_pair[int]();
  var ich2, och2 = mk_ioschannel_pair[int]();
```

Note, we have only created two channels here! But we have made two interfaces to the same channel, the first input, and the second output.

1.2.6 Connecting Devices with Channels

Now we can connect the devices to the channels:

```
  var p = producer och1;
  var t = transducer (ich1, och2);
  var c = consumer (ich2);
```

We have created procedure closures which bind the channel arguments to the procedures so that now the three closures all have the type `unit->void` which is required for the next step.

1.2.7 Spawning Fibres

Now we spawn active fibres from the coroutine closures:

```
  spawn_fthread p;
  spawn_fthread t;
  spawn_fthread c;
}
doit();
```

What we have done here is spawn three fibres which then communicate via the connected channels. The configuration in a series is called a pipeline and corresponds directly to functional composition.

1.2.8 Termination

Now you may wonder, how does it all end? What happens is that when the producer terminates by a procedural return which is called *suicide*. The transducer tries to read a value which is never going to come. The transducer is said to *starve*. The consumer also waits forever for a value from the transducer which is never going to come, because the transducer is starving, so the consumer also starves.

It is also possible for a coroutine to *block*. This happens when it tries to write a value which will never be read. Lets modify our example to see: an infinite production stream:

```
proc producer (out: %>int) () {
  var i = 0;
  while true do
    write (out, i);
    ++i;
  done
}
```

but a limited sample of data are printed:

```
proc consumer (inp: %<int) () {
  for i in 0..<10 do
    var y = read inp;
    println y;
  done
}
```

Now, the transducer blocks when the consumer terminates, and thus the producer blocks because the transducer has.

The astute programmer will have a number of questions! When a pre-emptive thread starves or blocks, it is a serious problem. Have we made a mistake with our fibres?

Here, you start on your journey to a major paradigm shift! Blockage and starvation are not an error with coroutines, its normal, expected, and desirable! This is, in fact, the main way that we organise termination!

Before I can explain this, however, I have to back step a bit!

1.3 Garbage Collection and Reachability

Felix runs a garbage collector similar to most functional programming languages. What a collector does is maintain a specified set of root objects, and finds all the objects to which there is a pointer in one of the roots. It then expands the set to include all the objects for which there is a pointer in one of those objects, and so on. If an object A has a pointer to an object B, we say B is directly reachable from A. If B then has a pointer to C, then C is said to be reachable from A, by first visiting B. The complete set of objects reachable from the designated roots is the transitive closure of the reachability relation. The other objects which are not reachable are garbage and are deleted. There's no way to refer to such an object, since there are no pointers to it in the roots, or any object reachable from the root.

Now, the secret of Felix coroutines is as follows: when you spawn a coroutine, the resulting fibre is reachable by the system, but it is NOT reachable from the caller. There is no "thread-id" returned when a coroutine is spawned, if you want to communicate with it you have to use a channel. The coroutine is named, the fibre spawned, however, is *anonymous*.

Now what happens is very simple but you will have to concentrate to get it! Coroutines passed channels can reach the channel. Any procedure which stores the channel can reach the channel. But the channel is an object and initially it can't reach anything. However when a coroutine performs I/O on the channel it can be suspended. If a read is done, and there is not yet a matching write, the fibre is suspended by adding it to the channel. Now the channel can reach the fibre. At the same time the system *forgets* the fibre. The system keeps a list of active fibres, but a suspended fibre is not active so it is forgotten.

A read operation is matched by a write, and a write operation is matched by a read. When a matching I/O operation is performed on a channel it means that the other operation that matches it has already been performed by another fibre. In this case, the channel forgets that fibre, and *both* that fibre and the one performing the matching operation become active and reachable by the system.

1.4 Indeterminacy

When fibres synchronise with matching I/O operations, both become active but only one actually starts executing. Which one is *indeterminate*. Felix always runs the reader first, but in the abstract semantics you are not allowed to know that. Indeterminacy is as close to concurrency as we can get with a sequential program and its vital not only for optimisation, but to ensure the programmer does not get bogged down depending on implementation details.

So now that you understand reachability, you will begin to understand what happens when a fibre starves. Provided there is no active fibre which can reach

the channel, then since the only object which can reach the fibre is the channel, which is unreachable, the starving fibre is also unreachable. So it is garbage collected!

Note *very carefully* that it is *absolutely essential* that channels only be reachable by fibres that wil use them. Go back and look carefully at the `doit` procedure:

```
proc doit () {
  var ich1, och1 = mk_ioschannel_pair[int]();
  var ich2, och2 = mk_ioschannel_pair[int]();
  var p = producer och1;
  var t = transducer (ich1, och2);
  var c = consumer (ich2);
  spawn_fthread p;
  spawn_fthread t;
  spawn_fthread c;
}
doit();
```

The four channel end points are known to this procedure, so whilst this procedure is active, those channels are reachable. Indeed the three closures `p,t,c` are bound to these channels, and the procedure knows them too. So the fibres spawned by this procedure may be reachable whilst the procedure itself is active.

Now, when you spawn a fibre, what happens? Does the spawned fibre run immediately, or does the spawning procedure continue?

Did you guess? In the abstract semantics, it is indeterminate! You're not allowed to design code that depends on which one runs first. In Felix, the spawned procedure runs first, but that's an implementation detail!

So what happens here is that sometime or other, the procedure will return, and the channels it could reach will no longer be reachable because the procedure's local data frame is no longer reachable.

And then, the procedure's data frame will be reaped by the collector, and, when the spawned fibres finally terminate, starve or block, they will also be reaped.

If you're getting the picture you may well wonder how the program as a whole terminates, and the answer is: in Felix the mainline is a coroutine! It is not a subroutine. In fact in Felix, all subroutines are, in the abstract, coroutines. The normal procedural subroutines are just coroutines that do not do channel I/O.

Chapter 2

Advantage of coroutines

You may wonder why bother with coroutines? What's wrong with ordinary functions and procedures?

The answer is: in the right context, functions and procedures are very useful. But they're a lot weaker than you think. Coroutines are to be treated as another technique, not a replacement for other techniques.

We will exhibit a critical case which shows beyond doubt that your conceptions about how great functional programming is are completely misplaced. Functional programming is great for functions but does not work so well when dealing with non termination or partial functions. In fact, it is so weak that the so called functional programming paradigm can be considered totally discreted along with object orientation.

These system has a shared fault: the subroutine. Subroutines involve a master slave relationship which skews your program design one way or another, and no way is natural. Coroutines fix this problem so you only use subroutines when they're natural. Coroutines provide a peer to peer relationship when that is the best way to do things.

2.1 Folds

The example I will use requires you to pretend that something simple could be more complicated and to envisage what that entails. I am going to use the classic functional programming function, the fold and show that functional programming is evil, and fold is perfect example of what is wrong with functional programming!

2.1.1 List Folds

First lets see a list in Felix:

```
union list[Element] =
  | Empty
  | Cons of Element * list[Element]
;
```

Now a top down, or left fold:

```
fun fold_left[Element, State]
  (acc: Element->State->State])
  (init: State)
  (ls: list[Element])
=>
  match ls with
  | Empty => init
  | Cons (head, tail) =>
    fold_left acc (acc head init) tail
;
```

I have written the routine in a functional style using recursion, it is in fact tail recursive. A right fold starts from the other end of the list and traditionally looks like this:

```
fun fold_right[Element, State]
  (acc: State->Element->State)
  (ls: list[Element])
  (init: State)
=>
  match ls with
  | Empty => init
  | Cons (head, tail) =>
    fold_right acc init (fold_right acc tail)
;
```

It is not tail recursion. Instead, we recurse down to the end of the list and fold the elements in to the result as we pop back up.

2.1.2 Tree Folds

A binary search tree has more useful orderings.

```
union tree [Element] =
  | Leaf
  | Node of Element * tree[Element] * tree[Element]
;
```

All tree visitors in a functional setting use a recursion, however the order of visiting elements is determined by when the client accumulator is called. Prefix order, or left most depth first is the easiest:

2.1.3 Pre-order Fold

This fold visits the deepest left most element first.

```
fun preorder [Element, State]
  (acc: Element -> State -> State)
  (init: State)
  (tr: tree[Element])
=>
  match tr with
  | Leaf => init
  | Node (elt, left, right) =>
    let lv = preorder acc init left in
    let v = acc elt lv in
    preorder acc v right
;
```

2.1.4 Post-order Fold

A post-order fold is similar, only it visits the right branch first:

```
fun postorder [Element, State]
  (acc: Element -> State -> State)
  (init: State)
  (tr: tree[Element])
=>
  match tr with
  | Leaf => init
  | Node (elt, left, right) =>
    let rv = postorder acc init right in
    let v = acc elt rv in
    postorder acc v left
;
```

2.1.5 Top down fold

This fold processes the element at the parent before the children:

```
fun midleftorder [Element, State]
  (acc: Element -> State -> State)
  (init: State)
  (tr: tree[Element])
=>
  match tr with
  | Leaf => init
  | Node (elt, left, right) =>
    let mv = acc elt init in
    let lv = midleft acc mv left in
    midleft acc lv right
;
```

This one visits the left child first.

2.1.6 Breadth first fold

To do!

2.2 Iterators

Another way to visit values in a data structures is to use iterators. We will show iterators corresponding to the folds above in a style similar to what would be required in C++, however we will use a single get method to get the next value which returns an option type, so that the end of the data stream can be detected. We'll also use a purely functional style.

2.2.1 Left List Iterator

The left visitor is quite easy, we use a list of the same type as the input as the state:

```
fun first_llit[Element]
  (ls: list[Element])
=>
  match ls with
  | Empty => None[Element], Empty[Element]
  | Cons (head, tail) => Some head, tail
;

fun next_llit[Element] => first_llit ls;
```

2.2.2 Right List Iterator

The bottom up iterator is harder:

```
fun first_rlit[Element]
  (ls: list[Element])
=>
  first_llit (rev ls)
;

fun next_rlit[Element] => first_llit ls;
```

The code is simple, it just uses the left iterator on a reversed list. There is an overhead in space and time which existed for the functional fold as well. The difference is, the iterator must maintain the state explicitly on the heap, whereas the fold uses the machine stack to hold pointers into the list implicitly.

2.2.3 Left Treee Iterator

Now the real fun starts! How must our iterator work?

```
union Todo[Element] =
  | Value of Element
  | Tree of tree[Element]
;
typedef zipper[Element] = list[Todo[Element]];

obj Iterata[Element] (tr: tree[Element]) =
{
  var path = Empty[Todo[Element]];
  setup (tr);

  proc setup (t: tree[Element]) {
    match t with
    | Leaf => return;
    | Node (elt, left, right) =>
      path = Cons (Tree right, path);
      path = Cons (Value elt, path);
      setup left;
    endmatch;
  }

  method gen next () => {
    match path with
    | Empty => return None[Element];
    | Cons (Value v, tail) =>
      path = tail;
      return Some v;
    | Cons (Tree t, tail) =>
      path = tail;
      setup t;
      return next();
    endmatch;
  }
}
```

As you can see, this is considerably more complicated than the corresponding fold. In fact, it took me a couple of minutes to write the fold and many hours until I figured out how to write the iterator. It is basically the control inverse of the fold: it is the fold turned inside out.

Control inversion is a key concept. The fold function is a master which calls the client function as a slave. The iterator, on the other hand, is a slave function called by the client, which is the master.

The iterator above reveals the true nature of the data required by a preorder tree visitation: the zipper above represents a type which can hold the state. In the fold the use of the machine stack, recursion, and local variables hides the

zipper: it uses the machine stack to couple the program counter with the local data. The iterator cannot do that, since it loses the stack each call. Instead it manually maintains its own stack, the path value.

The iterator above is only an input iterator. We can translate the mutations to produce a forward iterator by using a monadic form:

```
fun setup(t: tree[Element]) (path: zipper[Element]) =>
  match t with
  | Leaf => path
  | Node (elt, left, right) =>
    let path2 = Cons (Tree right, path) in
    let path3 = Cons (Value elt, path2) in
    setup left path3
;

fun next (path: zipper[Element]) =>
  match path with
  | Empty => None[Element], Empty[Todo[Element]]
  | Cons (Value v, tail) =>
    Some v, tail
  | Cons (Tree tr, tail) =>
    let path2 = setup tail path in
    next path2
;
```

2.3 Zippers

For any inductive data type, there is a related type known as a zipper. A zipper is basically a path in the tree. It can be thought of as the original data type with a hole in it representing the location of the current visitor, that is, a way to cut off a branch forming a subtree and a tree with a missing branch.

If we use the pure form of a tree it is a given by the formula:

```
typedef tree[T] = 1 + T * tree[T] * tree[T];
```

where + is the infix operator for an anonymous union or sum type, and 1 is the unit type. This has the form of a polynomial

$$1 + TX^2$$

which has the derivative

$$2TX$$

or

```
typedef zipper[T] = bool * T * tree[T]
```

In this form the boolean value is used to decide whether to process the value term next, or the right tree branch. In the zipper I presented, I split these two cases up into two list components `Value` and `Tree` and put them in the desired order in advance, because that was easier to understand than presenting both with a selector. If the selector is false, we process the value and set it to true, if it is true we process the tree and then discard the zipper node: in my implementation the value is first on the stack, then comes the tree, so the correct order is obtain by simply popping each element from the zipper as it is processed.

What's critical again is that the fold and iterator both maintain the same data that the zipper specifies, although the encoding is quite different, the fold maintaining it entirely implicitly.

Functional code is not always easier! If you consider a breadth first ordering, then both the functional and iterator versions must manually maintain some state. A simple functional breadth first fold could be done with a recursive descent and a depth limit, so all the values at a given depth are processed, then all the values one level deeper, and so on. Arranging termination is not entirely trivial, but could be done by simply calculating the maximum depth, again using a recursive descent. The fold itself would pass over intermediate level nodes several times, indeed in the degerate case of a list the top node would be scanned N times, for a list of length N, and the overall performance would be quadratic.

A better algorithm could avoid rescanning by keeping track better. This can be done by accumulating a list of all the children, reversing it and scanning it processing its values, and building up the list for the next pass. This handles termination correctly however the performance advantage is questionable since the list has to be constructed which takes time and uses up space: when the list is discarded it loads up the garbage collector. A simple recursive descent in a balanced binary tree only doubles the cost, because the number of ancestors of a set of siblings is always exactly equal to the number of siblings minus one.

2.4 The Client Code

I have shown the fold and control inverted fold, the iterator, so you can see clearly that in general the fold is superior because it is simpler. It can use recursion and call the client code wherever and whenever it wants. The iterator form is at a severe disadvantage because it is a callback or slave subroutine.

It is vital to understand that this imbalance is not because functional programming is better: the iterator form has a monadic functional equivalent, which is just as complicated as the procedural form, if not more so: the advantage of

the functional form is that it produces a forward iterator, the disadvantage is that the client code must maintain the state, which in our examples is the path representing the derivative.

What you must now see is that if you use a master fold, your client code has a problem: it is a slave: the client argument of a fold function is a callback, and if it is to do something complicated, it must manually maintain state. It cannot use local variables, recursion, or the machine stack.

On the other hand, the iterator form rules supreme for the client code. It can use recursion and the machine stack, and simply call the iterator whenever and wherever it wants for the next value. If you're using the functional forward iterator, you have to pass the state value around, but this is equivalent to being able to access the iterator object in the procedural form.

You will need some imagination to see that whilst in general state is required with both the fold and iterator methods, the use of manually constructed data structures on the heap may be necessary in the functional form if the linear machine stack is inadequate. Iterator clients are not necessarily trivial!

However the superiority of the iterator form .. for the client programmer .. is easy to demonstrate unequivocably by considering a really simple and very old algorithm: the merge.

A merge takes two sorted lists and merges them into a single sorted list. The algorithm is simple: look at the head of each list and pop off the smallest element, push it onto another list. Keep going until both lists are empty and reverse the result list.

Simple, and easy to make purely functional but there's a minor problem. Its simple if the client code can choose which list to pop. So its simple if you have two iterators!

In fact with iterators .. the same algorithm works, even if the data is coming from a tree instead of a list, because the iterator is converting the data structure into a value stream in all cases.

So how do you do a merge using a pair of folds?

Er .. well you can't. The iterators are clearly and unequivocably superior. The writer of a fold has an advantage. The client has a disadvantage. In this case, the disadvantage is a killer.

The way to do this is run the two folds separately to make two lists and then use then to feed your client code. If your language is lazily evaluated this is not necessarily inefficient but now we're exposing a well known major weakness of functional programming: the performance of your algorithms depends heavily on your compiler and run time implementation.

It is indeed interesting that lazy evaluation may allow the suspension of two folds over two trees whose client code produces two lists which are then consumed by

a merge. [More needs to be said here!]

2.5 Solving the problem with coroutines

It is critical at this point to understand there is a problem. Fold simply doesn't work, the client code is too hard to write! On the other hand iterator client code is easy to write, but the iterators are too hard to write!

Neither method is any good! Both suffer from the same problem: slavery! Slave subroutines do not implicitly retain state on the machine stack synchronised with control flow as masters do. Masters are better!

What's the answer? It's pretty obvious, we need two stacks, and we need two masters. But you cannot do that with mere subroutines. You cannot do it with traditional procedural code nor with functional code. Procedures and functions are both subroutines.

Since you read the title of this paper you already know the answer: coroutines. Coroutines cooperate as peers. They're constructed as if they're masters so both the visitor of the data structure and client code are easy. In fact we shall soon see, the data structure driver code in coroutine form looks exactly like the superior functional fold, and the client code looks exactly like the superior iterator client code. With coroutines, we can capture the best of both worlds!

2.6 Coding Visitors with coroutines

We're now going to attack the tree fold problem using coroutines. Recall the depth first functional routine:

```
fun preorder [Element, State]
  (acc: Element -> State -> State)
  (init: State)
  (tr: tree[Element])
=>
  match tr with
  | Leaf => init
  | Node (elt, left, right) =>
    let lv = preorder acc init left in
    let v = acc elt lv in
    preorder acc v right
;
```

Now see the coroutine, this is a full working example. First some test data:

```
union tree [Element] =
  | Leaf
  | Node of Element * tree[Element] * tree[Element]
;

var t =
  Node (10,
    Node (5,
      Node (1, Leaf[int], Leaf[int]),
      Node (7, Leaf[int], Leaf[int])
    ),
    Node (15,
      Node (12, Leaf[int], Node (13,Leaf[int], Leaf[int])),
      Node (17, Node (16, Leaf[int], Leaf[int]), Leaf[int])
    )
  )
;
```

and now the coroutine:

```
proc copreorder [Element]
 (out: %>Element)
 (tr: tree[Element])
{
  match tr with
  | Leaf => return;
  | Node (elt, left, right) =>
    copreorder out left;
    write (out, elt);
    copreorder out right;
  endmatch;
}
```

and now we need a way to check the results:

```
proc printer (ch: %<int) { println$ read ch; printer ch; }

begin
  var inp, out = mk_ioschannel_pair[int]();
  spawn_fthread { copreorder out t; };
  spawn_fthread { printer inp; };
end
```

2.7 Syntax Extensions

Felix has two syntax extensions designed to so coroutines are easier to use.

2.7.1 The `chip` definition

This extensions encourages a picture of coroutines as integrated circuits, even though that is not really accurate.

```
chip producer
  connector io
    pin out: %>int
{
  for i in 0..<10
    perform write (io.out, i);
}
chip transducer
  connector io
    pin inp: %<int
    pin out: %>int
{
  while true do
    var x = read io.inp;
    var y = x * x;
    write (io.out, y);
  done
}
chip consumer
  connector io
    pin inp: %<int
{
  while true do
    var y = read io.inp;
    println y;
  done
}
```

Here `connector` names an argument to the procedure of record type. The fields of the record and specified with the `pin` clause. You can have more than one connector phrase, each specifies a separate argument. Each `chip` has an aditional unit argument added automatically.

2.7.2 The `device` statement

You can write:

```
device x = y;
```

to construct a procedure closure of type `unit->void`. Actually, device is just a synonym for `var`, and is provided to make your look more like an electrical engineer than a software engineer.

2.7.3 The `circuit` statement

A `circuit` statement can be used to connect devices and pins. It is an executable statement!

```
circuit
  connect producer.out, transducer.inp
  connect transducer.out, consumer.inp
endcircuit
```

This makes a pipeline from the chips. The connecting channels are automatically created, as are the procedure closures required to make devices. The resulting devices are then spawned.

You can list any number of comma separated device/pin pairs in a connect clause. Felix finds the transitive closure of connections and makes a channel to connect all those pins together. The data type of all connected pins must be the same. If all are inputs or all are outputs, the compiler will issue a warning (but it is not an error!).

There is also another clause you can use in a circuit statement:

```
circuit
  wire ch to dev.pin
endcircuit
```

The `wire` clause allows you to connect a known channel to a device.

Chapter 3

Coroutine Semantics

3.1 Objects

A coroutine system consists of the following types of objects:

Scheduler A device to hold a set of active fibres and select one to be current.

Channels An object to support synchronisation and data transfer.

Fibres A thread of control which can be suspended and resumed.

Continuations An object representing the future of a coroutine.

3.1.1 Scheduler States

A scheduler is in one of two states:

Current The currently running scheduler

Suspended A scheduler for which the Running fibre is executing another scheduler.

3.1.2 Fibre States

Each fibre is in one of these states:

Running Exactly one fibre per scheduler is always running.

Active Fibes which are ready to run but not running on a particular scheduler.

Hungry Fibres suspended waiting for input on a channel.

Blocked Fibres suspended waiting to perform output on a channel.

3.1.3 Channel States

Each channel is in one of these states:

Empty There are no fibres associated with the channel.

Hungry A set of hungry fibres are waiting for input on the channel.

Blocked A set of blocked fibres are waiting to perform output on the channel.

3.2 Abstract State

3.2.1 State Data by Sets

A fibration system consists of

1. A set of fibres $\mathcal{F}$
2. A set of channels $\mathcal{C}$
3. An integer k
4. An indexed set of schedulers $\mathcal{S} = \{s_i\}$ for $i = 1$ to k

and the following relations:

1. for each $i = 1$ to k a pair $(R_i, \mathcal{A}_i)$ where R_i is a fibre and $\mathcal{A}_i$ is a set of fibres, these fibres being associated with scheduler s_i, R_i is the currently Running fibre of the scheduler, and $\mathcal{A}_i$ is the set of Active fibres;
2. for each channel c a set $\mathcal{H}_c$ of Hungry fibres and a set $\mathcal{B}_c$ of Blocked fibres, such that one of these sets is empty, if both sets are empty, the channel is said to be Empty, otherwise it is said to be Hungry or Blocked depending on whether the Hungry or Blocked set is nonempty;
3. A reachability relation to be described below

with the requirement that each fibre is in precisely one of the sets $\{R_i\}$, A_i, H_c or B_c.

We define the relation

$$H = \{(f, c) \mid f \in H_c\} \quad \text{Hunger} \tag{3.1}$$
$$B = \{(f, c) \mid f \in B_c\} \quad \text{Blockage} \tag{3.2}$$
$$\mathcal{F}_\mathcal{H} = \{f \mid \exists c.(f, c) \in \mathcal{H}\} \quad \text{Hungry Fibres} \tag{3.3}$$
$$\mathcal{F}_\mathcal{B} = \{f \mid \exists c.(f, c) \in \mathcal{B}\} \quad \text{Blocked Fibres} \tag{3.4}$$
$$\mathcal{C}_\mathcal{H} = \{c \mid \exists f.(f, c) \in \mathcal{H}\} \quad \text{Hungry Channels} \tag{3.5}$$
$$\mathcal{C}_\mathcal{B} = \{c \mid \exists f.(f, c) \in \mathcal{B}\} \quad \text{Blocked Channels} \tag{3.6}$$
$$\mathcal{E} = \{c \mid |\mathcal{H}_c = \mathcal{B}_c = \emptyset\} \quad \text{Empty Channels} \tag{3.7}$$

3.2.2 State Data by ML

Using an ML like description may make the state data easier to visualise.

```
scheduler =
  Run: fibre | NULL,
  Active: Set[fibre]

channel =
  | Empty
  | Hungry: NonemptySet[fibre]
  | Blocked: NonemptySet[fibre]

fibre = (current: continuation)

continuation =
  caller: continuation | NULL,
  PC: codeaddress,
  local: data
```

3.3 Operations

3.3.1 Spawn

The spawn operation takes as an argument a unit procedure and makes a closure thereof the initial continuation of a new fibre. Of the pair consisting of the currently running fibre (the spawner) and the new fibre (the spawnee) one will have Active state and the other will be Running. It is not specified which of the pair is Running.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{3.8}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{f\} \\ f, \mathcal{A}_k \cup \{R_s\} \end{cases} \tag{3.9}$$

where the choice between the two cases is indeterminate.

3.3.2 Run

The run operation is a subroutine. It increments k and creates a new scheduler s_k. The scheduler s_{k-1} is Suspended.

$$k \leftarrow k + 1 \tag{3.10}$$

It then takes as an argument a unit procedure and makes a closure thereof the initial continuation of a new fibre f and makes that the running fibre R_k of the new current scheduler. The set of active fibres A_k is set to $\emptyset$.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{3.11}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow f, \emptyset \tag{3.12}$$

The scheduler is then run as a subroutine. It returns when there is no running fibre, which implies also there are no active fibres left. k is then decremented, scheduler s_k again becomes Current, and the the current continuation of its running fibre resumes.

$$k \leftarrow k - 1 \tag{3.13}$$

3.3.3 Create channel

A function which creates a channel.

$$\mathcal{C} \leftarrow \mathcal{C} \cup \{c\} \tag{3.14}$$
$$\mathcal{E} \leftarrow \mathcal{E} \cup \{c\} \tag{3.15}$$

where c is a fresh channel.

3.3.4 Read

The read operation from fibre r takes as an argument a channel c.

1. If the channel is Empty, the Running fibre performing the read changes state to Hungry, the channel changes state to Hungry, and the fibre is associated with the channel.

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{3.16}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{3.17}$$

   If there are no active fibres, the program terminates, otherwise the scheduler selects an Active fibre and changes its state to Running. It is not specified which active fibre is chosen.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{3.18}$$

2. If the channel is Hungry, the Running fibre changes state to Hungry, and the fibre is associated with the channel.

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{3.19}$$
$$\tag{3.20}$$

   If there are no active fibres, the program terminates, otherwise the scheduler selects an Active fibre and changes its state to Running. It is not specified which active fibre is chosen.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if} A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A_k \end{cases} \tag{3.21}$$

3. If the channel is Blocked, one of the associated Blocked fibres w is selected, and dissociated from the channel.

$$\mathcal{B} \leftarrow \mathcal{B} \setminus (w, c) \tag{3.22}$$

   Of these two fibres, one is changed to state Active and the other to Running. It is not specified which fibre is chosen to be Running.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{w\} \\ w, \mathcal{A}_k \cup \{R\} \end{cases} \tag{3.23}$$

   The value supplied to the write operation of the Blocked fibre will be pass to the Hungry fibre when it transitions to Running state.

3.3.5 Write

The write operation performed by fibre w takes two arguments, a channel and a value to be written.

1. If the channel is Empty, the Running fibre performing the write changes state to Blocked, the channel changes state to Blocked, and the fibre is associated with the channel.

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{3.24}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{3.25}$$

If there are no active fibres, the program terminates, otherwise the scheduler selects an Active fibre and changes its state to Running. It is not specified which active fibre is chosen.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{3.26}$$

2. If the channel is Blocked, the Running fibre changes state to Blocked, and the fibre is associated with the channel.

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{3.27}$$

$$\tag{3.28}$$

If there are no active fibres, the program terminates, otherwise the scheduler selects an Active fibre and changes its state to Running. It is not specified which active fibre is chosen.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{3.29}$$

3. If the channel is Hungry, one of the associated Hungry fibres r is selected, and dissociated from the channel.

$$\mathcal{H} \leftarrow \mathcal{H} \setminus (r, c) \tag{3.30}$$

Of these two fibres, one is changed to state Active and the other to Running. It is not specified which fibre is chosen to be Running.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{r\} \\ r, \mathcal{A}_k \cup \{R\} \end{cases} \tag{3.31}$$

The value supplied by the write operation of the Blocked fibre will be pass to the Hungry fibre when it transitions to Running state.

3.3.6 Reachability

The Running, and, each Active fibre and its associated call chain of continuations are deemed to be Reachable.

If a channel is known to reachable fibre, it is also reachable. A channel may be known because its address is stored in the local data of a continuation of a

fibre, or, it is reachable via some object which can be reached from local data. The exact rules are programming language dependent.

Each fibre associated with a reachable channel is reachable.

The transitive closure of the reachability relation consists of a closed, finite, collection or channels and fibres which are reachable.

Unreachable fibres and channels are automtically garbage collected.

3.3.7 Elimination

Fibres and channels are eliminated when they are no longer reachable.

A fibre may become unreachable in three ways.

Suicide

A fibre for which the initial continuation returns is said to be dead, and becomes unreachable. If there are no longer any Active fibres, the program returns, otherwise the scheduler picks one Active fibre and changes its state to Running.

Starvation

A fibre in the Hungry state becomes unreachable when the channel on which it is waiting becomes unreachable.

Blockage

A fibre in the Blocked state becomes unreachable when the channel on which it is waiting becomes unreachable.

3.4 LiveLock

If a fibre is Hungry (or Blocked) on a reachable channel but no future Running fibre will write (or read) that channel, the fibre is said to be livelocked. The fibre will never proceed but it cannot be removed from the system because it is reachable via the channel.

A livelock is considered to transition to a deadlock if the channel becomes unreachable, in which case the fibre will becomes unreachable and is said to die through Starvation (or Blockage), disolving the deadlock. In other words, fibres cannot deadlock.

3.5 Fibre Structure

Each fibre consists of a single current continuation. Each continuation may have an associated continuation known as its caller. The initial continuation of a freshly spawned fibre has no caller.

The closure of the caller relation leads to a linear sequence of continuations starting with the current continuation and ending with the initial continuation of a freshly spawned fibre.

The main program consists of an initially Running fibre with a specified initial continuation.

Continuations have the usual operations of a procedure. They may return, call another procedure, spawn new fibres, create channels, and read and write channels, as well as the other usual operations of a procedure in a general purpose programming language.

A continuation is reachable if it is the current continuation of a reachable fibre, or the caller of a reachable continuation.

A continuation is formed by calling a procedure, which causes a data frame to be constructed which contains the return address of the caller, parameters and local variables of the procedure, and a program counter containing the current locus of control (code address) within the procedure. The program counter is initially set to the specified entry point of the procedure.

A coroutine is a procedure which directly or indirectly performs channel I/O. Coroutines may be called by other coroutines, but not by procedures or functions. Instead, a coroutine may be spawned by a procedure, or run by a procedure or function. This creates a fibre which hosts the created continuation.

Note: the set of fibres and channels created directly or indirectly by a run subroutine called inside a function should be isolated from all other fibres and channels to ensure the function has no side-effects.

3.6 Continuation Structure

3.6.1 Continuation Data

A continuation has associated with it the following data:

caller Another continuation of the same fibre which is suspended until this continuation returns.

data frame Sometimes called the stack frame, contains local variables the continuation may access.

program counter A location in the program code representing the current point of this continuations execution or suspension

3.6.2 Continuation operations

The current continuation of a fibre executes a wide range of operations including channel I/O, spawning new fibres, calling a procedure, and returning.

call Calling a procedure creates a new continuation with its program counter set at the procedure entry point, and a fresh data frame. The new continuation becomes the current continuation, the current continuation suspends. The new continuations caller field is set to the caller. The current continuation program counter is set to the pointer after the call instruction.

The effect is push an entry onto the fibres continuation chain.

return Returning from the current continuation causes the owning fibres current continuation to be set to the current continuations caller, if one exists, or the fibre to be marked Dead if there is no caller. Execution of the suspended caller continues at its program counter.

The effect is to pop an entry off the fibre's continuation chain.

read/write Channel I/O suspends the current continuation of a fibre until a matching operation from another fibre synchronises with it. A read is matched by a write, and a write is matched by a read.

By the rules of state change, channel I/O should be viewed as performing a peer to peer neutral exchange of control: the current fibre becomes suspended without losing its position and hands control to another fibre. Later, control is handed back and the fibre continues.

Coroutine based systems, therefore, operate by repeated exchanges of control accompanied by data transfers in a direction independent of the control flow, which sets coroutines aside from functions.

3.7 Events

Each state transfer of the fibration system may be considered an event. However the key events are

- spawning
- suicide
- entry to a read operation
- return from a read operation

- entry to a write operation
- return from a write operation

I/O synchronisation consists of suspension on entry to a read or write operation, and simultaneously release of suspension, or resumption, on matching write or read.

I/O suspension occurs when a fibre becomes Hungry or Blocked, and resumption when it becomes Running or Active.

Fibrated systems are characterised by a simple rule: events are totally ordered. The order may not be determinate.

3.8 Control Type

The control type of a coroutine is defined as follows. We assume the coroutine is spawned as a fibre, and each and every read request is satisfied by a random value of the legal input type. Write requests are also satisfied. We cojoin entry and return from read into a single read event, and entry and return from write into a single write event, since we are only interested in the behaviour of the fibre.

The sequence of all possible events which the fibre may exhibit is the coroutines control type. Note, the control type is a property of the coroutine (procedure).

3.9 Encoding Control Types

In general, the control type of a coroutine can be quite complex. However for special cases, a simple encoding can be given.

3.9.1 One shots

A one-shot coutine is one that exhibits a bounded number of events before suiciding. The three most common one shots are:

Value: type W A coroutine which writes a single value to a channel and then exits.

Result: type R A coroutine which reads a single value to from channel and then exits.

Function: type RW A couroutine which reads one value from a channel, calculates an answer, writes that down a channel and then exits.

3.9.2 Continuous devices

A continuous coroutine is one which does not exit. It can therefore terminate only by starvation or blockage. The three most common kinds of such devices are

Source: type W+ Writes a continuous stream of values to a channel.

Sink: type R+ Reads a continuous stream of values from a channel.

Transducer Reads and writes.

Because the sequence of events is a stream, we may use convenient notations to describe control types. If possible, a regular expression will be used. Sometimes, a grammar will be required. In other cases there is no simple notation for the behaviour of a coroutine.

We will use postfix + for repetition.

3.9.3 Transducer Types

A transducer which read a value, write a value, then loops back and repeats is called a *functional transducer*, it may be given the type (RW)+.

In a functional language, a partial function has no natural encoding. There are two common solutions. The first is to return an option type, say Some v, if there is a result, or None if there is not. This solution involves modifying the codomain. The other solution is to restrict the domain so that the subroutine is a function.

Coroutines, however, represent partial functions naturally. If a value is read for which there is no result, none is written! The type of a *partial function transducer* is therefore given by ((R+)W)+, in other words multiple reads may occur for each write. Note that two writes may not occur in succession.

This type may also be applied to many other coroutines, for example the list filter higher order function.

3.9.4 Duality

Coroutines are dual to functions. The core difference is that they operate in time not space. Thus, in the dual space a spatial product type becomes a temporal sequence.

Coroutines are ideal for processing streams. Whereas function code cannot construct streams without laziness, and cannot deconstruct them without eagerness, coroutines are neither eager nor lazy.

One may view an eager functional application as driving a value into a function to get a result, and a lazy application as pulling a value into a function. Pushing value implies eagerly evaluating it, pulling implies the value is calculated on demand.

Coroutine simultaneously push and pull values across channels and so eliminate the evaluation model dichotomy that plagues functional programming. This coherence does not come for free: it is replaced by indeterminate event ordering.

3.10 Composition

By far the biggest advantage of coroutine modelling is the ultimate flexibility of composition. Coroutines provide far better modularity and reusability than functions, but this comes at the price of complexity. You will observe considerably more housekeeping is required to compose coroutines than procedures or functions, because, simply, there are more way to compose them.

A collection of coroutines can be regarded as black boxes resembling chips on a circuit board, with the wires connecting pins representing channels. So instead of using variables and binding constructions, we can construct more or less arbitrary networks.

3.10.1 Pipelines

The simplest kind of composition is the pipeline. It is a sequence of transduces wired together with the output of one transducer connected by a channel to the input of the next.

If the pipeline consists entirely of transducers is is an open pipeline. If there is a source at one end and a sink at the other it is a closed pipeline. Partially open pipelines can also exist.

The composition of two transducers has a type dependent on the left and right transducer types.

With a functional transducer, you would expect the composition of (R1W1)+ with (R2W2)+ to be (R1W2)+ but this is not the case!

Consider, the left transducer performs R1, then W1, then right performs R2. At this point it is not determinate whether left or right proceeds. If left proceeds, we have R1 again, then W1. then right proceeds and performs W2 before coming back to read R2, and what happens next is again indeterminate. The sequence is therefore R1, W1/R2, R1, W1/R2 which shows R1 can be read twice before W2 is observed. We have written w/r here to indicate synchronised events which are abstracted away when describing the observable behaviour of the composite.

Clearly, (R1?R1W2?W2)+ contains the set of possible event sequences, but then (R1+R2+)+ contains it, and therefore the set of possible event sequences as well. So we should seek the most precise, or *principal* type of the composite.

We can calculate the type from the operational semantics. At any point in time, the system must be in one of a finite number of states. Where we have indeterminacy, the transitions out of a given state are not fully specified. The result is clearly a non-deterministic finite state automaton.

We must observe, such an automaton corresponds to (one or more) larger deterministic finite state automata. This is an important result because it has practical implications: it means we can pick a DFA and use it to optimise away abstracted synchronisation points. In other words, we build a fast model of the system by inlining and using shared variables instead of channels, and then eliminate the variables by functional composition.

This is the primary reason we insist on indeterminate behaviour: it allows composition to be subject to a reduction calculus.

3.11 Felix Implementation

The following functions and procedures are provided in Felix:

```
spawn_fthread: (1 -> 0) -> 0;
run: (1 -> 0) -> 0;
mk_ioschannel_pair[T]: 1 -> ischannel[T] * oschannel[T];
read[T]: ischannel[T] -> T
write[T]: oschannel[T] * T -> 0
```

In the abstract, channels are bidirectional and untyped. However we will restrict our attention to channels typed to support either read (ischannel) or write (oschannel) of a value of a fixed data type.

The following shorthand types are available:

```
%<T    ischannel[T]
%>T    oschannel[T]
```

More advanced typing exploiting channel capabilities are discussed later.

Simple example program:

```
proc demo () {
  var inp, out = mk_ioschannel_pair[int]();

  proc source () {
    for i in 1..10 perform write (out,i);
  }

  proc sink () {
    while true do
      var j = read inp;
      println$ j;
    done
  }

  spawn_fthread source;
  spawn_fthread sink;
}
demo();
```

In this program, we create a channel with an input and output end typed to transfer an int. The source coroutine writes the integers from 1 through to 10 inclusive to the write end of the channel, the sink coroutine reads integers from the channel and prints them.

The main fibre calls the demo procedure which launches two fibres with initial continuations the closures of the source and sink procedures.

When demo returns, the main fibre's current continuation no longer knows the channel, so the channel is not reachable from the main fibre.

The source coroutine returns after sending 10 integers to the sink via the channel. When a fibre no longer has a current continuation, returning to the non-existent caller causes the fibre to no longer have a legal state. This is known as suicide.

After the sink has read the last value, it becomes permanently Hungry. The sink procedure dies by starvation.

All fibres which die do so either by suicide, starvation, or blockage. Dead fibres will be reaped by the garbage collector provided they're unreachable. It is important for the creator of fibres and their connecting channels to forget the channels to ensure this occurs.

Unlike typical pre-emptive threading systems, deadlock is not an error. However a lock up which should lead to reaping of fibres but which fails to do so because they remain reachable is univerally an error. This is known as a livelock: it leads to zombie fibres.

This usually occurs because some other fibre is statically capable of resolving

the lockup, but does not do so dynamically. To prevent livelocks, variables holding channel values to which no I/O will occur dynamically should also go out of scope.