Programming with Coroutines

John Skaller

June 25, 2017

Contents

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.

The most significant picture of the advantages of coroutines is thus: in a subroutine based language there is a single machine stack. By machine stack, I mean that there is an important *implicit* coupling of control flow and local variables. In the abstract, a subroutine call passes a continuation of the caller to the callee which is saved along with local variables the callee allocates, so that the local variables can be discarded when the final result is calculated, and then

passed to the continuation. This technique may be called *structured programming*. With coroutines, the picture is simple: each fibre of control has its own stack. Communication via channels exchanges data and control between stacks.

Coroutines therefore leverage control and data coupling in a much more powerful and flexible manner than mere functions, reducing the need for state to be preserved on the heap, thereby making it easier to construct and reason about programs.

For complex applications, the heap is always required.

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. The type of variable `out` is denoted `%>int` which is actually short hand for `ochannel[int]`, which is an output channel on which values of type `int` may be written.

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
}
```

Here, the type of variable `inp` is denoted `%<int` which is actually short hand for `iochannel[int]`, which is an input channel from which values of type `int` may be read.

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
}
```

Each of these components is a coroutine because it is a procedure which may perform, directly or indirectly, I/O on one or more synchronous channels.

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. They 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 provides modularity and encapsulation and allows one to reason locally. This is a vital information hiding property which is also possessed by pure functions, where it is known as *functional abstraction*.

They key idea of functional abstraction is that an approximation of the function semantics is represented by the function type. For example the functions `modulus` and `argument`

```
fun modulus (z:dcomplex) : double => sqrt (z.x^2 + z.y^2);
fun argument (z:dcomplex) : double => arctan2 (z.y, z.x);
```

both have type `dcomplex -> double`, so the type is only an approximation which forgets some details of the function, which is the usual meaning of abstract. Never-the-less the type is useful to allow the type checker to prevent calling these functions on an integer, but more importantly it allows for higher order functions:

```
fun map (x:list[dcomplex]) (f:dcomplex->double) =>
  match x with
  | Empty => Empty[double]
  | Cons (head, tail) => Cons (f head, map f tail)
  endmatch
;
```

This function will take a list of `dcomplex` and apply either `modulus` or `argument` or any other function with the type `dcomplex->double` to the list to produce a list of `double` safely: the point is that this map function does not need to know the full semantics of the argument to which the parameter `f` is bound, only that it has the correct type.

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 `1->0`, where 1 is also named as `unit` and 0 is named as `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.

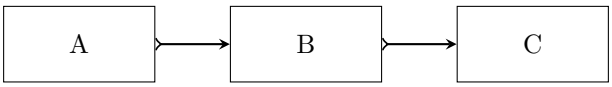

Figure 1.1: Reachability

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

```
#if FLX_CGOTO
  #define FLX_LOCAL_LABEL_VARIABLE_TYPE void*
  #define FLX_PC_DECL void *pc;
#else
  #define FLX_PC_DECL int pc;
  #define FLX_LOCAL_LABEL_VARIABLE_TYPE int
#endif
```

Figure 1.2: Code Address

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.

A more detailed explanation follows. A formal definition of the precise execution semantics is given in chapter 6 Coroutine Semantics.

1.4 Execution Model

When Felix starts your program, the machine stack is reachable, and so any object with a pointer on the machine stack is reachable. In addition, your initial mainline code is implicitly a coroutine, which is spawned automatically creating a fibre object which contains a pointer to the fibre's initial continuation. The fibre is running, so it is also reachable.

All your top level variables are stored in an object called the *thread frame*. Continuation objects contain a pointer to the thread frame so that the procedure can access the global variables. A continuation object is also known as the procedure *activation record* or *data frame*, or, historically, its *stack frame*. As well as a pointer to the thread frame, a continuation object also contains a pointer to the most recent activation record of its ancestors, in fact the thread frame may be consider a universal ancestor.

Vitally, continuation objects also contain a value known as the program counter. This value is the current location at which the continuation is executing, it always points into the code of the procedure the frame represents. When a subroutine is called, the program counter is set to the statement after the subroutine call, a new continuation is created for the subroutine, its program counter is set to the first statement, and a back pointer to the caller continuation is stored. The back pointer, together with the program counter of the caller, are together known as the subroutine *return address*. It represents the continuation which the subroutine resumes when the subroutine itself is complete.

Then the current continuation of the fibre is changed to the new continuation.

When a return statement is executed, the backpointer to the caller is stored

```
struct con_t
{
  FLX_PC_DECL               // interior program counter
  struct _uctor_ *p_svc;    // service request

  virtual con_t *resume()=0;// computation step
  con_t * _caller;          // return address
};
```

Figure 1.3: Continuation base

```
struct fthread_t
{
  con_t *cc;                // current continuation
};
```

Figure 1.4: Fibre

```
struct slist_node_t {
  slist_node_t *next;
  fthread_t *data;
};

struct slist_t {
  gc::generic::gc_profile_t *gcp; // garbage collector
  struct slist_node_t *head;
};

struct schannel_t
{
  slist_t *waiting_to_read;  // fthreads waiting for a writer
  slist_t *waiting_to_write; // fthreads waiting for a reader
};
```

Figure 1.5: Synchronous Channel

into the fibre object, and execution continues with the caller at the statement after the subroutine call.

Thus, the continuations form a singly linked list which operates like a stack. The continuation objects are heap allocated and the data structure is known as a *spaghetti stack*. In principle, a continuation also has a pointer to the most recent activation record of its parent, which has a pointer to its parent, until the list terminates with the thread frame, so there are **two** interleaved lists here: one representing the call chain, and one representing the static nesting structure: that is what make it a spaghetti stack. In Felix, pointers to all the ancestors are stored in the continuation object to improve access time to ancestoral variables, at the cost of passing them all to each child (however the optimiser does lots of magic).

Each of the frame pointers mentioned is known to the garbage collector and so a single reachable running procedure defines a transitive closure of reachable objects. Note that in addition, any variable containing a Felix pointer obtained from a manual heap allocation ensures the heap object is reachable if the variable is in a reachable frame. In addition, any reachable pointer which points anywhere into an object ensures the object is reachable. If the pointer is not to the first byte, it is called an *interior pointer*. Note that in Felix a pointer "one past the end" of an object does not make the object reachable!

Now all this explains, technically, something easy to state loosely: if you can access an object it is reachable. In addition if the *system* can access the object it is reachable.

Now what I have described so far does not explain fibres. The currently running fibre, and all those deemed active are reachable by the system. When the currently running fibre performs an unmatched synchronous channel I/O operation, either a read or a write, it is added to the channel's list of suspended fibres and is removed from the set of fibre the system can reach directly. So the fibre can now only be reached from the channel. So it will be garbage unless another active fibre can reach the channel. After all since the I/O operation is unmatched, if another fibre can't see the channel, there is no fibre that can satisfy the I/O request.

When a fibre is suspeneded by a read or write operation, the program counter of its current continuation is set to the statement after the I/O operation. If the operation is later matched, the address of the data being transmitted is transfered from the writer to the reader, and the two fibres both made active so that they will continue at the statement after the I/O operation.

What is vital to realise now is that each fibre has its own spaghetti stack. So when control is exchanged from one fibre to another:

control exchange is effected by stack swapping

Of course, we mean the heap allocated spaghetti stacks. You can swap machine stacks too: this is done by the host operating system scheduler and the entities

being context switched are known as threads. The swaps are premptive, and several stacks can be running at once if you have a multi-core CPU. Pre-emptive threads are much harder to use than fibres, and the context switches are much more expensive. They are greatly overused in many programs for purpose of obtaining control inversion because the host language is deficient and does not support coroutines. This deficiency is shared by almost all production and research programming systems!

1.5 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?

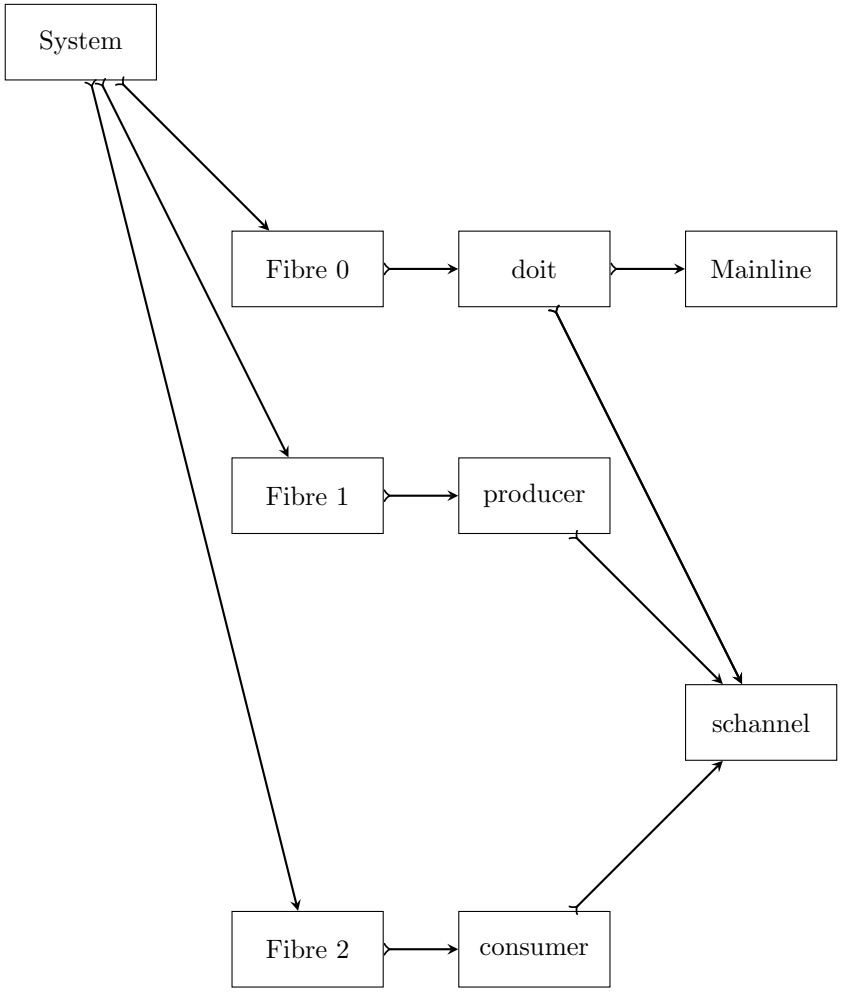

Figure 1.6: Reachability: After Spawning

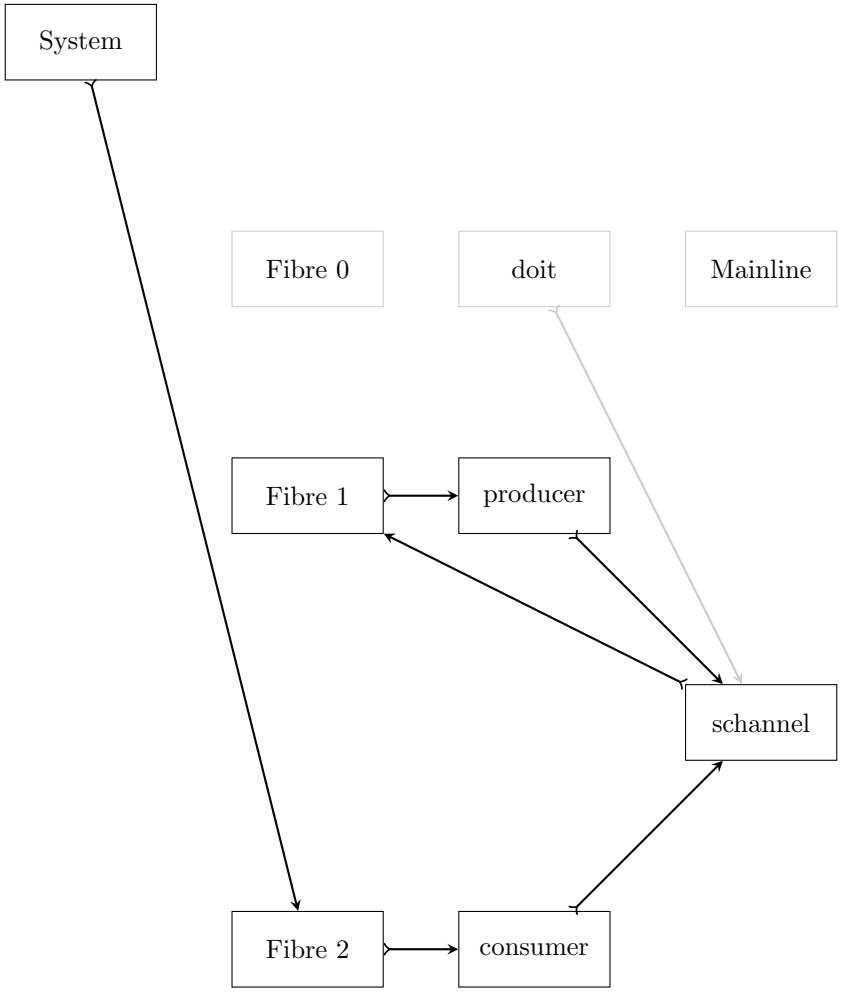

Figure 1.7: Reachability: Mainline completed, after Write, before Read

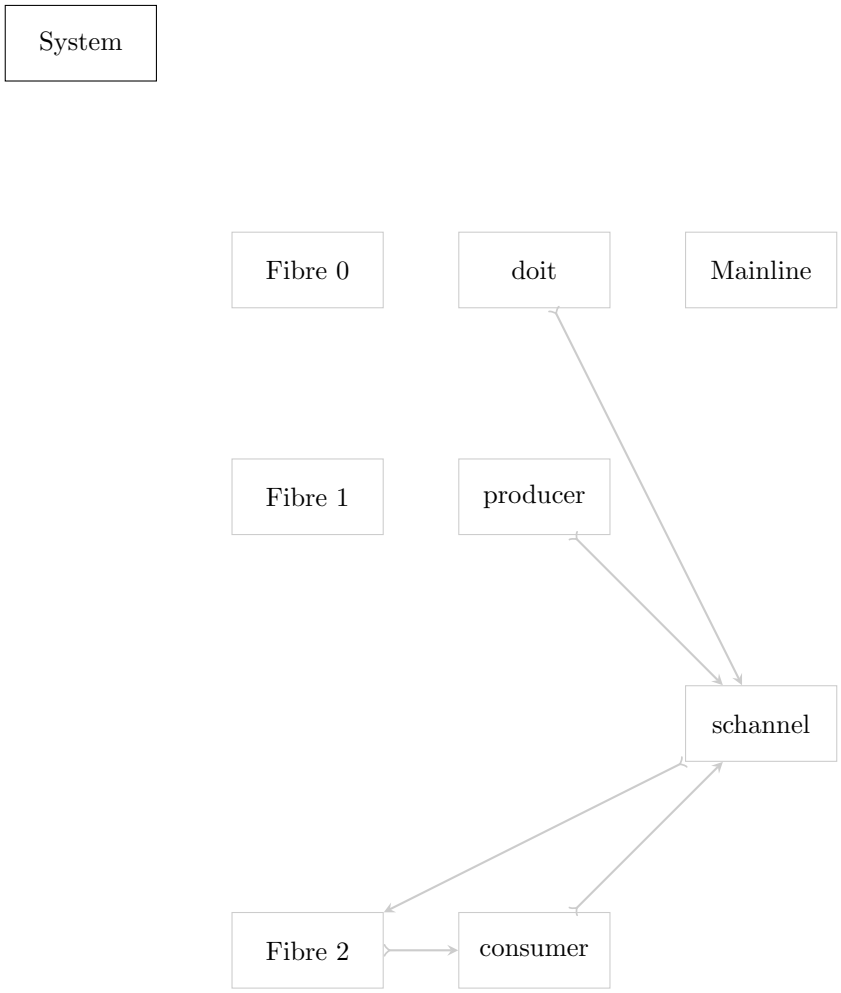

Figure 1.8: Reachability: Producer completed, Consumer starved, Program finished

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

Coroutine Basics

2.1 Syntax Extensions

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

2.1.1 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. The signatures of the three chips above are:

```
producer: (out: %out) -> 1 -> 0
transducer: (inp: %<int, out: %out) -> 1 -> 0
consumer: (inp: %<int) -> 1 -> 0
```

Note that this syntax uses a record type for the connector, whereas our original coroutines used a plain type for one parameter and a tuple for the two required for the transducer.

2.1.2 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.

2.1.3 The `circuit` statement

The `connect` clause

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

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

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

Figure 2.1: Simple circuit connection

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!).

The `wire` clause

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

Core Components

Every system needs a library!

3.1 Base Components

3.1.1 Blockers

Here is a device to use when you have to connected a writer to a channel, but want it to be unconnected.

```
chip writeblock[T]
  connector io
    pin inp : %<T
{
}
```

And the corresponding reader:

```
chip readblock[T]
  connector io
    pin out: %>T
{
}
```

These coroutines suicide immediately, so a writer is blocked, or a reader starved, respectively.

3.1.2 Universals

This chip reads input forever but ignores it.

```
chip sink[T]
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    C_hack::ignore (x);
  done
}
```

Whereas this chip writes a fixed value forever. It is the analogue of a value or constant function:

```
chip source[T] (a:T)
  connector io
    pin out: %>T
{
  while true perform write (io.out, a);
}
```

3.1.3 Adaptors

Two key adaptors provide *lifts*:

```
chip source_from_list[T] (a:list[T])
  connector io
    pin out: %>T
{
  for y in a perform write (io.out,y);
}

chip bound_source_from_list[T] (a:list[T])
  connector io
    pin out: %>opt[T]
{
  for y in a perform write (io.out,Some y);
  while true perform write None[T];
}
```

A lift is a way to go from the imperative/function model of the world into the coroutine/stream model. These two chips lift a list into a stream.

It is absolutely vital to understand how these two lifts differ. The first lift is a pure lift which simply starves any read trying to go past the end of the list. There is no terminal value to tell the reader the list has ended. Notice a finite number of values is written by the first device, equal to the number in the list. This is a *finite stream*.

The second device generates an infinite stream by embedding a finite list in its head using an option type. The tail of the stream is an infinite list of None's. The None values are terminators which act to bound the list.

I will warn now, to understand how to use the first device requires a paradigm shift. Having things drop dead without any warning seems difficult to manage if you're used to dealing with inductive data types in a functional setting. We will see later, however, that it is natural.

Next, we have a basic adaptor for a function.

```
chip function[D,C] (f:D->C)
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    var y = f x;
    write (io.out, y);
  done
}
```

This device is an example of the generic category of a *transducer*.

3.1.4 Drops

A drop is a way to get out of the coroutine/stream model back to your imperative functional model:

```
chip procedure[D] (p:D->0)
  connector io
    pin inp: %<D
{
  while true do
    var x = read io.inp;
    p x;
  done
}
```

The device is a sink which will typically create side effects for each value read.

A special case is a list drop:

```
chip sink_to_list[T] (p: &list[T])
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    p <- Cons (x,*p);
  done
}
```

Now, we need an example to show how to use this drop!

```
include "std/control/chips";
open BaseChips;

var output = Empty[int];

device s = source_from_list ([1,2,3,4]);
device tr = function (fun (x:int)=>x*x);
device d = sink_to_list &output;
run  {
  circuit
    connect s.out, tr.inp
    connect tr.out, d.inp
  endcircuit
};
println$ output;
```

The critical thing to note is the `run` procedure. It spawns its argument procedure as the initial fibre of a new fibre scheduler, and then waits until that scheduler terminates due to a lack of active fibres.

So *within* the fibre system, we cannot detect the end of the list, but from outside, we can detect it indirectly by the fact that our circuit is no longer active.

The `run` procedure first lifts out of the current procedural/imperative mode into fibrated stream mode, waits until it completes, and then drops back to procedural mode. In other words it interfaces the two modes.

`run` can be used in a procedure, in a coroutine, or in a function. Run, in effect, creates and pushes a scheduler on a scheduler stack, waits until it completes, and then pops back to the current scheduler.

3.1.5 Avoiding lockup

To avoid some cases of lockup we provide the buffer device:

```
chip buffer [T]
  connector io
    pin inp: %<T
    pin out: %>T
{
  while true do
    var x = read io.inp;
    write (io.out, x);
  done
}
```

You can see this is a just a copy operation and is a special case of the `function` chip, which uses the identity function. However in a fibrated setting, `buffer` is not semantically a no operation.

Here's an example:

```
include "std/control/chips";
open BaseChips;

chip out2
  connector io
    pin oa: %>int
    pin ob: %>int
{
  write (io.oa, 11);
  write (io.ob, 42);
}

chip in2
  connector io
    pin ia: %<int
    pin ib: %<int
{
   var a = read a;
   var b = read b;
   println $ a - b;
}

// WOOPS, lock up!
//circuit
//   connect out2.ob, in2.ia
//   connect out2.oa, in2.ib
//endcircuit

device abuf = buffer;
device bbuf = buffer;
circuit
   connect out2.ob, bufb.inp
   connect out2.oa, bufa.inp
   connect in2.a, bufa.out
   connect in2.b, bufb.out
endcircuit
```

This is a classic deadlock for threads. The writer writes a first then b, then reader reads b first, then a. Adding the buffers removes the ordering dependency. I added two buffers, although in this case only one is required. Can you figure out which two pins have to connected via a buffer?

Here is another useful chip, it copies a single value from input to output then suicides. It is generally useful to insert into pipelines that would otherwise be continuous when you only want a one shot operation.

```
chip oneshot [T]
  connector io
    pin inp: %<T
    pin out: %>T
{
  write (io.out, read io.inp);
}
```

Finally here are some convenience types:

```
typedef iopair_t[D,C] = (inp: %<D, out: %>C);

// source
typedef ochip_t[T] = (out: %>T) -> 1 -> 0;

// transducer
typedef iochip_t[D,C] = iopair_t[D,C] -> 1 -> 0;

// sink
typedef ichip_t[T] = (inp: %<T) -> 1 -> 0;
```

which specify the type of three commonly used chips.

Chapter 4

Pipelines

One of the most basic control structures you can build with coroutines is the *pipeline*. This is a series connection of transducers, the output of the left one of a pair connected to the input of the right one. A pipeline of transducers is said to be an *open pipeline*.

If a source is connected to the left end, and a sink to the right end, it is a closed pipeline.

An open pipeline is semantically equivalent to a transucer with additional buffering. A pipline closed on the left is a source, and a pipeline closed on the right is a sink. The syntax —-¿ is parsed to pipe (a,b). We add overloads for chips with pins named io.inp, io.out.

Here are the binary combinators:

This chip connects two transducers to form a new transducer. Note, since we use the `circuit` statement, the pair of component coroutines are actually spawned as fibres.

```
chip pipe[T,U,V] (a:iochip_t[T,U],b:iochip_t[U,V])
 connector io
   pin inp: %<T
   pin out: %>V
{
  circuit
    connect a.out,b.inp
    wire io.inp to a.inp
    wire io.out to b.out
  endcircuit
}
```

Here we connect a source to a transducer to make a new source:

```
chip pipe[T,U] (a:ochip_t[T],b:iochip_t[T,U])
 connector io
   pin out: %>U
{
  circuit
    connect a.out,b.inp
    wire io.out to b.out
  endcircuit
}
```

Here, a transducer is connected to a sink to form a new sink.

```
chip pipe[T,U] (a:iochip_t[T,U],b:ichip_t[U])
 connector io
   pin inp: %<T
{
  circuit
    connect a.out,b.inp
    wire io.inp to a.inp
  endcircuit
}
```

Finally, connecting a source to a sink results in a closed pipeline. Closed pipelines are equivalent in some sense to subroutines in that they can only be observed for their side effects.

```
// source to sink
proc pipe[T] (a:ochip_t[T],b:ichip_t[T]) ()
{
  circuit
    connect a.out,b.inp
  endcircuit
}
```

Note carefully, this last operator is a procedure not a chip!

Felix provides the infix symbol `|->` for the `pipe` operator. An example of use, we can say:

```
producer |-> transducer |-> consumer;
```

given the chips of 2.1.1 instead of the circuit statemnent 2.1.

4.0.1 The lift functor

Pipelining is associative up to buffering. The exact structures spawned may differ and the order of execution may differ, but the ordering always conforms to the abstract semantics.

There is a mapping between function compositions and pipelines, and this mapping is structure preserving. Given a sequence of functions of types suitable for composition

$$f_1, f_2, f_3 \dots$$

then writing Φ for the `function` chip, we have

$$\Phi(f_1 \odot f_2 \odot f_3 \dots) \cong \Phi f_1 \mapsto \Phi f_2 \mapsto \Phi f_3 \dots)$$

where $\odot$ is reverse function composition.

In other words, it is structure preserving, and thus a categorical *functor*. It is called the *lift* functor because it lifts functional code into cofunctional code, that is, functional stuff is lifted into semantically equivalent coroutine based code. You can also drop any pipeline to a function composition, so the two systems are isomorphic.

The key point, which we are yet to demonstrate, is that pipelines are not the only kind of circuits you can make. Cofunctional programming *subsumes* functional programming. Its more flexible and more powerful.

4.0.2 Linear Flow

Linear flow circuits are an extension of the pipeline concept which allows data to flow from sources to sinks in an acyclic network. Lets look at an example with two sources:

```
device A = source_from_list ([1,2,3]);
device B = source_from_list ([5,6,7]);
chip add
  connector io
    a: %<int
    b: %>int
    sum: %>int
{
  while true do
    var a = read io.a;
    var b = read io.b;
    write (io.sum, a + b);
  done
}
circuit
  connect A.out, add.a
  connect B.out, add.b
  connect add.sum, consumer.inp
endciruit
```

where we have used our original consumer to print the results. There is an important thing to observe here: the order in which our `add` chip reads it input does not matter *in this case* because it is connected to two *independent* sources.

You can probably see that given any binary operators represented as chips, we can construct a calculate with a tree like structure to perform the calculation.

```
chip sub
  connector io
    a: %<int
    b: %>int
    diff: %>int
{
  while true do
    var a = read io.a;
    var b = read io.b;
    write (io.diff, a - b);
  done
}
```

If you are familiar with functional programming concepts, you may ask whether these functions are eagerly or lazily evaluated. Eager evaluation means the arguments are evaluated first, before the function is called, so if such an evaluation fails to terminate, the function call never happens, and we can say that the whole application fails to terminate.

With lazy evaluation, the arguments are only evaluated when they're actually needed. So if an argument which would be nonterminating is not actually needed, the function application can still succeed.

Because coroutines provide explicit control flow, the evaluation strategy depends on the way you write the coroutine. To put this another way, there is no built in preference for either eager or lazy evaluation. We will demonstrate by showing an important operator written two different ways. First the eager variant:

```
chip eagerconditional
  connector io
    pin condition: %<bool
    pin trueval: %<int
    pin falseval: %<int
    pin result: %>int
{
  var c = read io.condition;
  var t = read io.trueval;
  var f = read io.falseval;
  write (io.result, if c then t else f);
}
```

and now the lazy variant:

```
chip lazyconditional
  connector io
    pin condition: %<bool
    pin trueval: %<int
    pin falseval: %<int
    pin result: %>int
{
  var c = read io.condition;
  if c do
    var t = read io.trueval;
    write (io.result, t);
  else
    var f = read io.falseval;
    write (io.result, t);
  done
}
```

If the eager chip starves on either the read of the true value or the false value, then any reader of the result also starves, no matter what is read for the condition. However the lazy chip only ever reads the value it is required to output, and so only starves if the the read on that channel starves. If it doesn't not, then it doesn't matter if a read on the other channel starves, because we never actually read it.

Another interesting chip is this one:

```
chip choose
  connector io
    pin condition: %<bool
    pin value: <%int
    pin truecont: %>int
    pin falsecont: %>int
{
  var c = read io.condition;
  var v = io.value;
  if c do
    write (io.truecont, v);
  else
    write (io.falsecont, v);
  done
}
```

This is a very important chip to understand! What it does is read a condition and a value and write that value down one of two channels, depending on the condition.

At the other end of the two outputs there may well be two different chips reading the result, one to handle each of the two conditions. So this chip is

like a conditional goto chip, or a switch, in that it choses how the rest of the program will proceed by selecting a data path. Whichever path is chosen, the continuation suspended at the end of the channel will continue execution. So passing an output channel to a chip is an abstract way of passing a continuation.

I say abstract because the actual chip which resumes control on reading a value from the channel is dependent entirely on how the circuit is connected. It doesn't depend on the actual channel passed directly, but what is connected to the other end.

Critically, the `choose` chip enforces lazy evaluation because only one of the channels is written to, what's connected to the other end will only be activated if its input channel is selected for the write. In particular I want you, the reader, to see that channels are not merely ways to send data around, rather, they're ways to transmit *control*. In particular, networks of connected chips have a shape called a *control structure*.

So now we have looked at extensions to the pipeline concept in which we have chips with multiple inputs and outputs, and we are going to demonstrate how to handle the partial function division:

```
chip divide
  connector io
    pin numerator: %<int
    pin denominator: %<int
    pin quotient: %>int
    pin divisionbyzero: %>int
{
  var n = read io.numerator;
  var d = read io.denominator;
  if d == 0 do
     write (io.divisionbyzero, numerator);
  else
     write (io.quotient, numerator/denominator);
  done
}
```

This is an important chip because it shows how to handle a partial function correctly, by providing an error channel.

Imagine we have to compute the formula:

$$\frac{x+y}{x-y}+1$$

We can use this:

```
var x = 1;
var y = 1;
device xc = x.source;
device yc = y.source;
device one = 1.source;

device add1 = add;
device add2 = add;

chip error
  connector io
    pin inp: %>int
{
  var x = read io.inp;
  println "Division of " + x.str + " by zero";
}

circuit
  connect add1.a, xc.out
  connect add1.b, yc.out
  connect sub.a, xc.out
  connect sub.b, yc.out
  connect div.numerator, add1.sum
  connect div.denominator, sub.diff
  connect div.quotient, add2.a
  connect one.out, add2.b
  connect add.sum, consumer.inp
  connect div.divisionbyzero, error.inp
endciruit
```

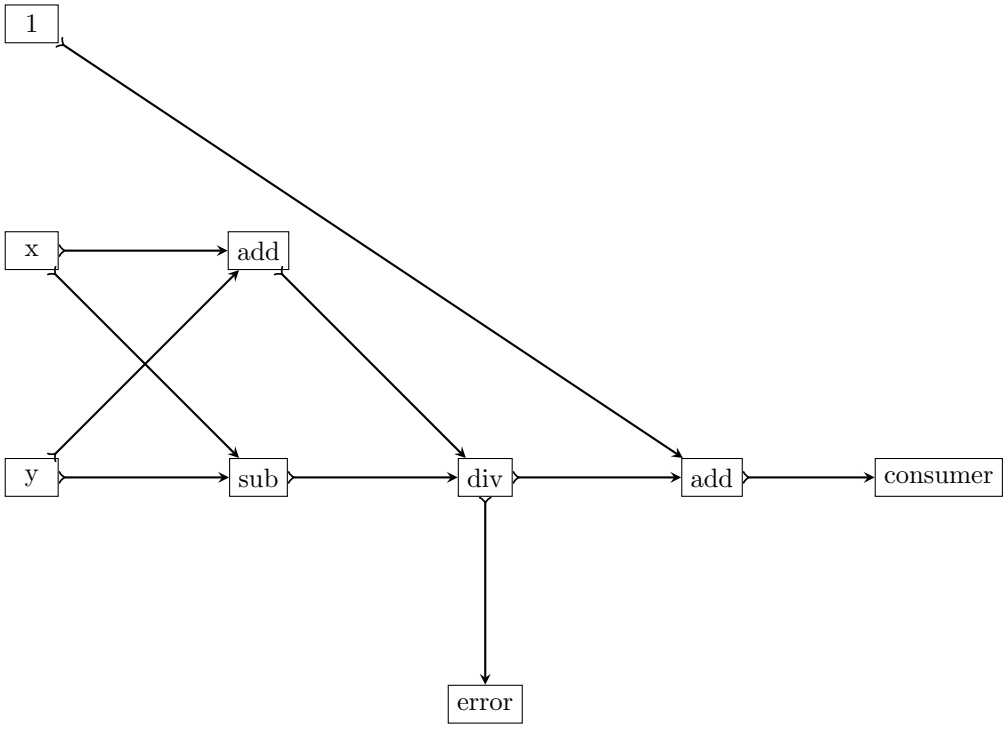

Figure 4.1: Flow in simple formula

Chapter 5

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.

5.1 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!

5.1.1 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) =>
    acc (fold_right acc tail init) head
;
```

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.

5.1.2 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:

5.1.3 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 v = acc elt lv in
    let lv = preorder acc init left in
    preorder acc v right
;
```

5.1.4 In-order Fold

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

5.1.5 Post-order Fold

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

5.1.6 Breadth First Fold

This is much harder to do efficiently. A simple routine is not so hard, lets start with a descent which folds a single level, it returns the folded value and a flag which tells if that level is populated:

```
fun bfn[Element, State]
  (acc: Element -> State -> State)
  (init: State)
  (tr: tree[Element])
  (n:int)
=>
  match tr with
  | Leaf => init, false
  | Node (elt, left, right) =>
    if n > 0 then
      let lv,lflag = bfn acc init left (n - 1) in
      let rv, rflag = bfn acc lv right (n - 1) in
      rv, lflag or rflag
    else
      acc elt init, true
    endif
;
```

The cost of this algorithm is roughly as follows: there are 2^k elements in level k where k starts at 0, assuming a fully populated balanced tree of sufficient depth, so we have to scan $\Sigma_{k=0}^{n} 2^k$ elements, including the target level n, but this is known to be just $2^{n+1} - 1$.

Now the idea is simply to fold level 1, then level 2, then level 3, etc, until the flag tells us we hit an entirely unpopulated level: this is easier to do imperatively

but we want a functional solution:

```
fun bfaux[Element, State]
  (acc: Element -> State -> State)
  (init: State)
  (tr: tree[Element])
  (n:int)
=>
  let v,flag = bfn acc init tr n in
  if flag then
     bfaux acc init tr (n + 1)
  else
    v
  endif
;

fun bf[Element, State]
  (acc: Element -> State -> State)
  (init: State)
  (tr: tree[Element])
=>
  bfaux acc init tr 0
;
```

Now the total cost of folding a fully populated balanced tree to level n is $\Sigma_{k=0}^{n+1}(2^k - 1)$ where we had to use $n + 1$ to account for the fully empty level below which has to be scanned for the termination check. We can throw out the -1 here, and the result is just 2^{n+2} which, suprisingly, is only four times the number of elements in the tree. The routine isn't as inefficient as you might think, and it has a major advantage of other routines: it uses no auxilliary storage, and the use of the stack is limited to some small multiple of the tree depth, which is insignificant if the tree is reasonable well shaped.

5.2 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.

5.2.1 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])
:
  opt[Element] * list[Element]
=>
  match ls with
  | Empty => None[Element], Empty[Element]
  | Cons (head, tail) => Some head, tail
;

fun next_llit[Element]
  (ls: list[Element])
:
  opt[Element] * list[Element]
=>
  first_llit ls
;
```

5.2.2 Right List Iterator

The bottom up iterator is harder:

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

fun next_rlit[Element]
  (ls: list[Element])
:
  opt[Element] * list[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.

5.2.3 Left Tree 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
;
```

5.3 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.

5.4 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!]

5.5 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!

5.6 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
```

Part I

Appendices

Chapter 6

Coroutine Semantics

6.1 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.

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.2 Abstract State

6.2.1 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

$$H = \{(f, c) \mid f \in H_c\} \quad \text{Hunger} \tag{6.1}$$

$$B = \{(f, c) \mid f \in B_c\} \quad \text{Blockage} \tag{6.2}$$

$$\mathcal{F}_{\mathcal{H}} = \{f \mid \exists c.(f, c) \in \mathcal{H}\} \quad \text{Hungry Fibres} \tag{6.3}$$

$$\mathcal{F}_{\mathcal{B}} = \{f \mid \exists c.(f, c) \in \mathcal{B}\} \quad \text{Blocked Fibres} \tag{6.4}$$

$$\mathcal{C}_{\mathcal{H}} = \{c \mid \exists f.(f, c) \in \mathcal{H}\} \quad \text{Hungry Channels} \tag{6.5}$$

$$\mathcal{C}_{\mathcal{B}} = \{c \mid \exists f.(f, c) \in \mathcal{B}\} \quad \text{Blocked Channels} \tag{6.6}$$

$$\mathcal{E} = \{c \mid |\mathcal{H}_c = \mathcal{B}_c = \emptyset\} \quad \text{Empty Channels} \tag{6.7}$$

6.2.2 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
```

6.3 Operations

6.3.1 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.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{6.8}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{f\} \\ f, \mathcal{A}_k \cup \{R_s\} \end{cases} \tag{6.9}$$

where the choice between the two cases is indeterminate.

6.3.2 Run

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

$$k \leftarrow k + 1 \tag{6.10}$$

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 $\emptyset$.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{6.11}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow f, \emptyset \tag{6.12}$$

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.

$$k \leftarrow k - 1 \tag{6.13}$$

6.3.3 Create channel

A function which creates a channel.

$$\mathcal{C} \leftarrow \mathcal{C} \cup \{c\} \tag{6.14}$$
$$\mathcal{E} \leftarrow \mathcal{E} \cup \{c\} \tag{6.15}$$

where c is a fresh channel.

6.3.4 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.

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{6.16}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{6.17}$$

   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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{6.18}$$

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

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{6.19}$$
$$\tag{6.20}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if} A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A_k \end{cases} \tag{6.21}$$

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

$$\mathcal{B} \leftarrow \mathcal{B} \setminus (w, c) \tag{6.22}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{w\} \\ w, \mathcal{A}_k \cup \{R\} \end{cases} \tag{6.23}$$

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

6.3.5 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.

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{6.24}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{6.25}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{6.26}$$

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

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{6.27}$$

(6.28)

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{6.29}$$

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

$$\mathcal{H} \leftarrow \mathcal{H} \setminus (r, c) \tag{6.30}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{r\} \\ r, \mathcal{A}_k \cup \{R\} \end{cases} \tag{6.31}$$

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

6.3.6 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.

6.3.7 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.

6.4 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.

6.5 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.

6.6 Continuation Structure

6.6.1 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

6.6.2 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.

6.7 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.

6.8 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).

6.9 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.

6.9.1 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.

6.9.2 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.

6.9.3 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.

6.9.4 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.

6.10 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.

6.10.1 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.

6.11 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 nonexistent 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.

Chapter 7

Listings

List of Listings