Programming with Coroutines

John Skaller

August 13, 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.

In the abstract theoretical sense, the fundamental property possessed by coroutines can be stated like this: within any thread, there exists some total ordering of all events. The ordering may not be determinate, but of any two events which occur, one definitely occurs before the other.

In addition, events associated with one fibre which occur between two synchronisation points, are never interleaved by events from another fibre of the same

thread. All interleaving must occur interior to the synchronisation point, that is, after it commences and before it completes. In other words, given a sequence of events from one fibre prior to a synchronisation point, and a sequence of event from another after a synchronisation point, all the events of each sequence occur before or after all the events of the other.

Premptive threads, on the other hand, allow arbitrary interleaving of each threads sequence of events, up to and after any shared synchronisation. Mutual exclusion locks provide serialisation, which is the default behaviour of coroutines.

Therefore, fibre based programming can proceed where general code may assume exclusive access to memory and other resources over all local time periods not bisected by a volutary synchronisation event; threads, on the other hand, can only assume exclusive access in the scope of a held mutex.

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.

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

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

Listing 1.1: Simple source

`oschannel[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
}
```

Listing 1.2: A simple transducer

Here, the type of variable `inp` is denoted `%<int` which is actually short hand for `ischannel[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
}
```

Listing 1.3: A simple sink

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]();
```

Listing 1.4: Construct channels

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);
```

Listing 1.5: Bind Channel Arguments

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();
```

Listing 1.6: Spawn Fibres

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
}
```

Listing 1.7: Infinite Source

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
}
```

Listing 1.8: Finite sink

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.

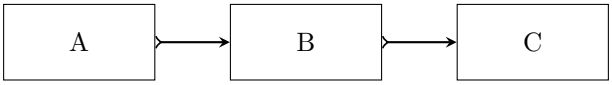

Figure 1.1: Reachability

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 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 12 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 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).

The stack of pointers to the most recent activation record of the lexical ancestors has a technical name, it is called a *display*.

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

```
#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
```

Listing 1.9: Code Address: C++ representation

The type of a code address depends on the capabilities of the compiler. The GNU compiler g++ for x86 and x86-64 family supports two features essential for our purposes, a *computed goto*, and the ability to find the machine address of a labelled position in the code from any other place in the code. This is achieved in Felix with a trick using some assembler stolen from Fergus Henderson's Mercury implementation. Although Clang also has a computed goto, addressing label C++ function is not possible because the assembler label hack is not supported. Clang requires any computed goto in a function can only jump into that function, and enforces this by disallowing the assembler label trick. Felix never jumps to a label other than one in the function doing the computed goto, but it could.

```
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
};
```

Listing 1.10: Continuation base: C++ representation

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

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

Listing 1.11: Fibre: C++ representation

```
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
};
```

Listing 1.12: Synchronous Channel

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?

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!

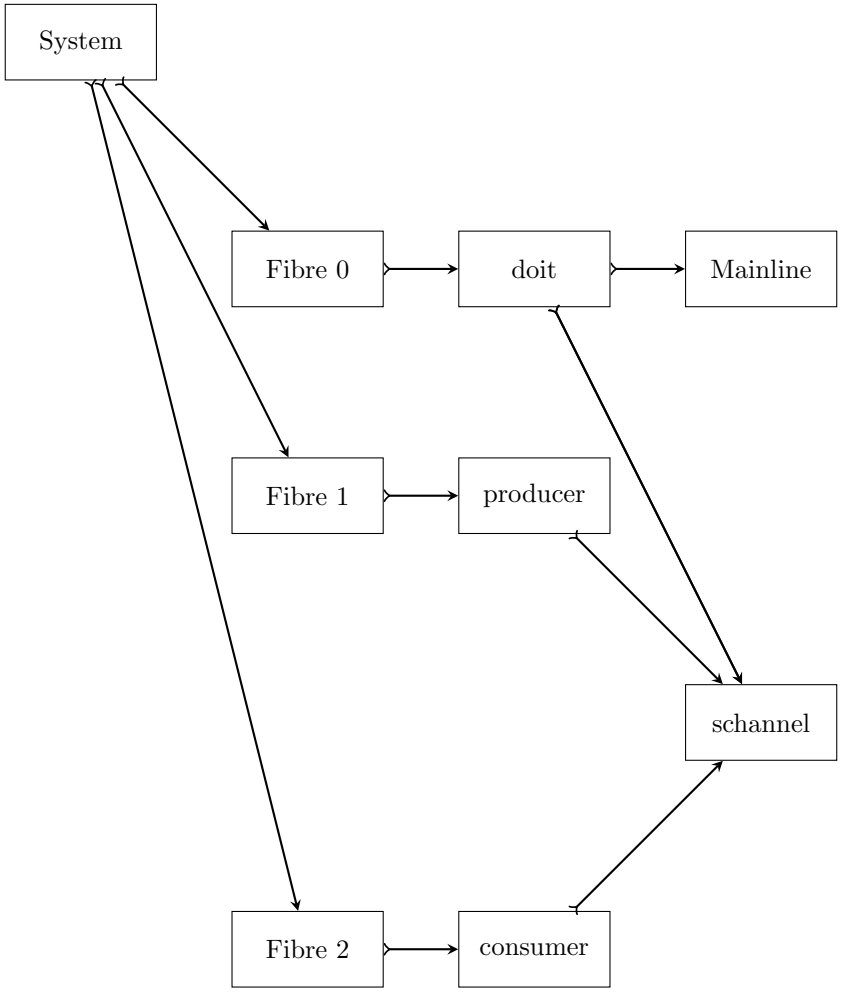

Figure 1.2: Reachability: After Spawning

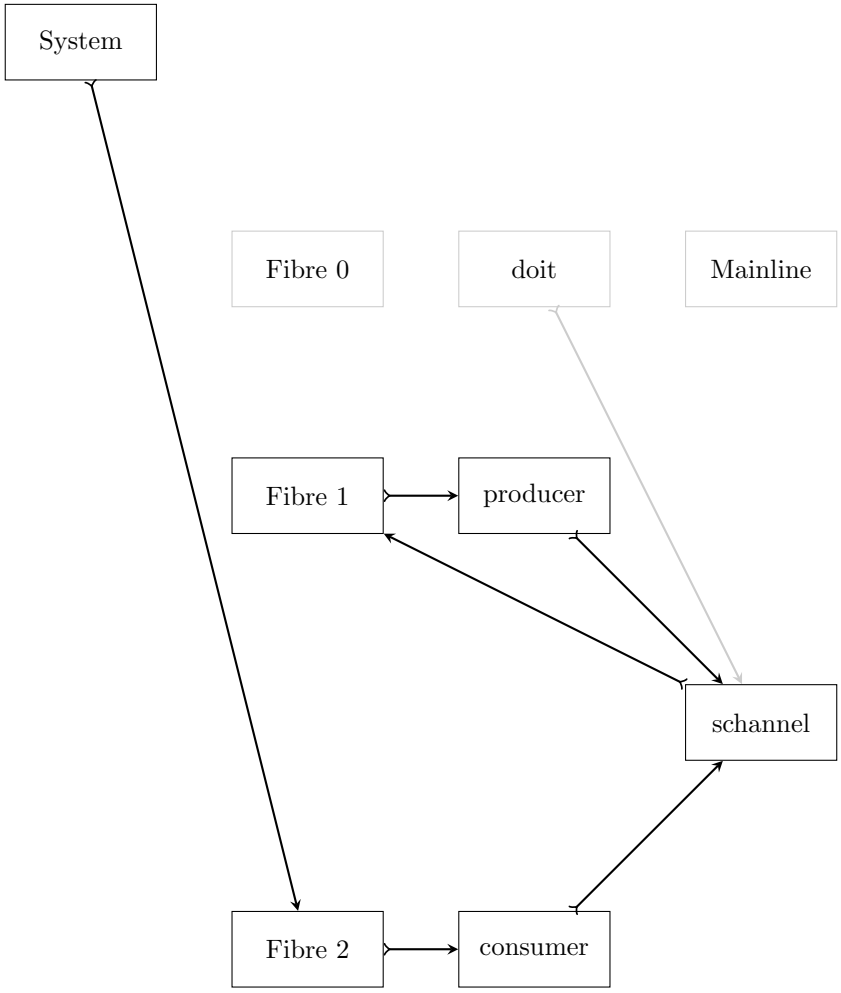

Figure 1.3: Reachability: Mainline completed, after Write, before Read

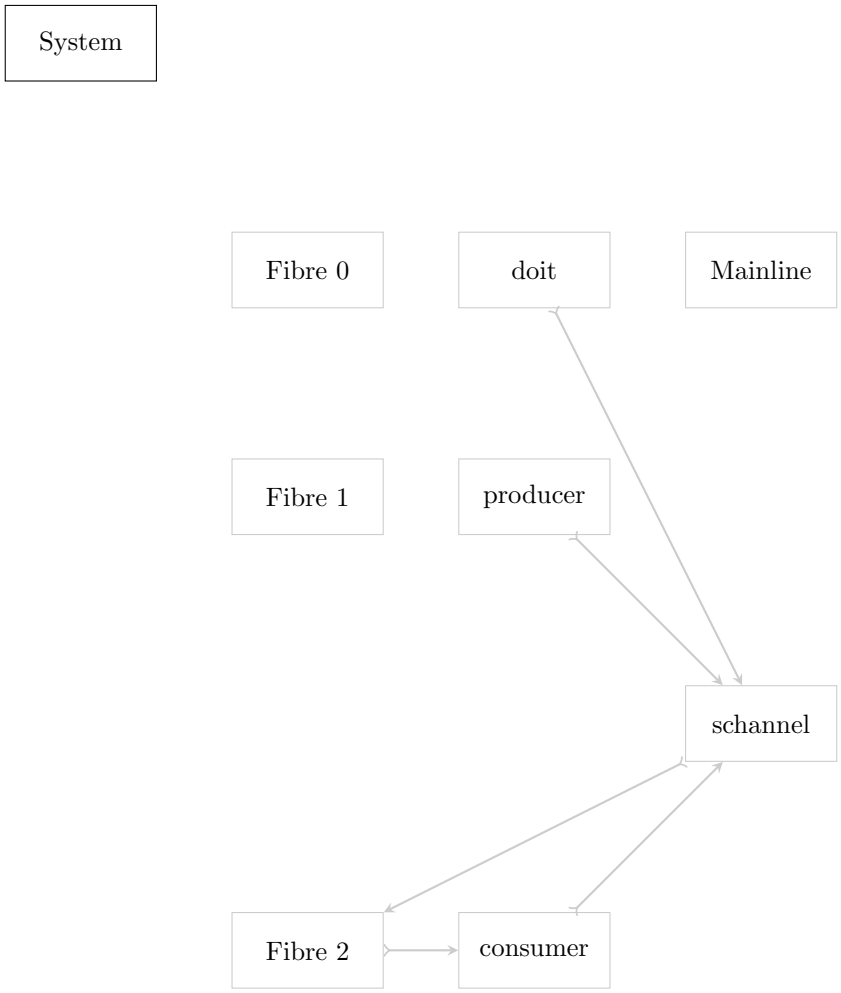

Figure 1.4: Reachability: Producer completed, Consumer starved, Program finished

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

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

2.1.1 The `chip` definition

This extension 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: %<int) -> 1 -> 0
transducer: (inp: %<int, out: %>int) -> 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 you 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!

The example 2.1 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.

Constraints

If you include a device in your circuit statement, it must have the type of a function accepting a record of synchronous channels, and returning a unit procedure. All the pins must be connected. The circuit statement ensures all connected devices are fully connected, and, it spawns all of these coroutines before completing. Thus, it simultaneously connects and activates a complete circuit, and does so whilst preventing the client from knowing any of the channels it constructs to perform the connections.

All pins connected together must either be input or output channels for the same type. If all the pins are output, or all the pins are input, a warning will be generated, but the compilation will proceed. Writing to any pin connected to channel with all other pins output necessarily blocks, if all are input, any read necessarily starves.

2.1.4 The `suicide` procedure

This library procedure causes the immediate termination of the current fibre.

```
suicide;
```

Figure 2.2: Suicide

2.2 Laziness

It is critical to understand *when* the code of a chip is run! Consider simple chip:

```
chip simple (n:int)
  connector io
    pin out: %>int
{
  for i in 0..<n perform write (io.out, i);
}
```

The type of this chip is:

```
int -> (out: %>int) -> 1 -> 0
```

that is, it is a function accepting an integer returning another function which accepts a record with field `out` returning a procedure which accepts the unit argument ().

Now if we say:

```
device  s1 = simple 10;
```

we have created an object with the parameter n fixed to 10. We say the object is a closure, because the parameter n is now fixed. The closure now has type

```
(out: %>int) -> 1 -> 0
```

As you can see, this is still a function!

We can again create a closure of that value by:

```
var s2 = s1 (out=ch);
```

where `ch` is a channel on which you can write an int. The closure `s2` has type:

```
1 -> 0
```

and we say the second parameter, namely `io`, is now bound.

The result, finally, is a procedure, and in fact a coroutine because it writes to a channel. The body of the code of the chip is the code of this procedure. So until the procedure is actually called or spawned, the code does not execute.

The chip is equivalent to this:

```
proc simple (n:int)
  (io: ( pin out: %>int) )
  ()
{
  for i in 0..<n perform write (io.out, i);
}
```

which in turn is equivalent to this:

```
fun simple (n:int)  =>
  fun (io: ( pin out: %>int) ) =>
    proc () {
      for i in 0..<n perform
        write (io.out, i);
    }
;
```

In other words, connecting a chip to channels is a matter of binding to create a closure by fixing the parameter representing the connector, or, attaching channels to the pins if you like: it can be done inside a function because the connected circuit is not activated: it hasn't yet been connected to a power source!

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 connect 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
}
```

The `C_hack::ignore` procedure is a trick which forces the read to occur. In Felix, if a variable is unused it is optimised away — along with its initialiser. In this case the initialiser is the read operation, which has a crucial and intended side effect so we don't want it optimised away. The ignore procedure throws away an unwanted result of a computation, without throwing away the computation itself. So the read will be performed, even though we don't care what the returned value is. In Felix it is always safe to throw away application of a function if the result is not used, since functions are not allowed to have any side effects. However read is a generator not a function.

The result of a generator must be used otherwise the generator application will be thrown away. This changes the program semantics. Felix does this deliberately, it is not a bug but a feature. A particularly important case is assigning a clock to a variable. Doing that can dynamically load the asynchronous I/O subsystem and start an event monitoring thread. But if the variable is not used, the clock isn't constructed, and the program may run without requiring the asynchronous I/O system. The system is loaded on demand, and the semantics ensure the demand propagates from the use of a feature requiring the system, the semantics also ensures resources aren't acquired *unless* they're needed.

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);
}
```

Here's a oneshot version of it:

```
chip value[T] (a:T)
  connector io
    pin out: %>T
{
  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/functional model of the world into the coroutine/stream model. These two chips lift a list into a stream. That is, it translates spatial, inductive, data into temporal, coinductive, codata.

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*.

And here is a basic adaptor for a procedure:

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

which is an example of the generic category *sink*.

3.1.4 Filter

Another useful chip is the filter, the first variant uses a condition on the input, if it passes then the function is applied to the input value to produce the output value, otherwise the value is ignored:

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

The second variant just uses a function with codomain of an option type:

```
chip filter[D,C] (f:D->opt[C])
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    match f x with
    | Some y => write (io.out, y);
    | None => ;
    endmatch;
  done
}
```

Another special filter is:

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

Filters are transducers whose equivalent mathematical entity is a partial function. This is relation for which not all values in the domain have a corresponding value in the codomain. You can upgrade a partial function to a function in several ways, the two most common being to restrict the domain to the values for which the partial function is defined, or, to extend the codomain with an error value which if mapped to, indicates the original partial function was not defined for that argument.

Unfortunately both these methods have a serious fault: they change the type of the partial function so it can no longer be composed with other functional entites. Some systems leave the type alone and throw an exception which turns out to be even more nasty.

Other systems systematically extend all functions to include the error value in both domains and codomains by using a monad. However whilst this restores composability of the extended operators, the monads themselves cannot be easily combined.

The correct solution for a functional language, and there is only one correct solution, is exactly what C does: nothing! It is up to the programmer, without help from the type system, to ensure that the partial function is not called with a value for which it is not defined. The set of defined values is sometimes represented by a precondition, which can be either just documentation, checked at run time, or in limited cases, checked at compile time.

Dependent typing systems extend the usual limitations of the compile time checks by requiring the client programmer supply suitable proof sketches.

Felix allows preconditions written using the `when` clause like this:

```
fun safe_divide (x:int, y:int when y != 0) => x/y;
```

The precondition is checked at run time.

3.2 Synchronising drops with `run`

An *drop* is the opposite of a lift. It translates codata into data, that it, it translates temporal ordering into a spatial ordering.

3.2.1 List drop

A special case of the `procedure` chip 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
}
```

It translates a stream into a list. That is, values arriving over time are stored, latest first, into a spatial data type.

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.

3.3 Symmetric Model

We have encountered now two operations, lifts and drops, which translate from space to time, and time to space, respectively. In general a source is a lifted value, and a sink is a dropped value.

In the functional world, we usually deal with three kinds of entities: a value of a type, which is an element of a set. We apply mysterious things called functions to values to obtain new values.

Values can also be fed into the computation representing the rest of the program, which is called a continuation. When you apply a function, the code to be done afterwards, which accepts the result, is the continuation of the application. In a procedural setting, the continuation can be thought of as being represented by the return address, which is invoked by the `return` statement. In fact the

return address is secretly passed to the procedure along with the argument, just so that the return statement know where to return to.

Another view is that calling a subroutine suspends the caller, invokes the callee, which, when finished, resumes the caller just after the call. With functions, the continuation is not, however complete, it must be passed the result of the function call before it can continue.

In this model of the world, we like to think about passing continuations to routines so they know what to do when their part of the job is done. When code is written such that continuations are explicitly passed around, it is called CPS or Continuation Passing Style.

Having learned about coroutines and how they work you may begin to see how synchronous channels act to pass continuations. A fibre suspended on a channel, waiting for matching I/O operation from another fibre is precisely a continuation.

3.4 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 be connected via a buffer?

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 transducer 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! It produces a closed chip, that is, one with no channels to connect. It is ready to call or spawn.

An example of use, we can say:

```
#(producer |-> transducer |-> consumer);
```

given the chips of 2.1.1 instead of the circuit statemnent 2.1. Note that this calls the pipeline, which causes it to begin execution sometime, not necessarily immediately. The effect is subtly different to this:

```
spawn_fthread (producer |-> transducer |-> consumer);
```

which spawns a fibre that itself then spawns the pipeline. If you want to force the pipeline to run immediately you have to do this:

```
run _fthread (producer |-> transducer |-> consumer);
```

This works because `run` is an ordinary subroutine specified to spawn its argument on a nested scheduler, and to return only when there are no active fibres on the scheduler, which occurs when the pipline is dead: all the fibres have suicided, starved or are blocked.

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

$$f_1, f_2, f_3 \dots \tag{4.1}$$

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

$$\Phi(f_1 \odot f_2 \odot f_3 \dots) \cong \Phi f_1 \mapsto \Phi f_2 \mapsto \Phi f_3 \dots \tag{4.2}$$

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 Pipeline from list

This chip allows dynamic construction of an open pipeline from a non-empty list of transducers.

```
chip pipeline_list[T] (a: list[iochip_t[T,T]])
  connector io
    pin inp: %<T
    pin out: %>T
{
  proc aux (lst:list[iochip_t[T,T]]) (inp: %<T) {
    match lst with
    | h1 ! h2 ! tail =>
      var inchan,outchan = mk_ioschannel_pair[T]();
      spawn_fthread$  h1 (inp=inp, out=outchan);
      aux (h2!tail) inchan;
    | h1 ! _ =>
      spawn_fthread$  h1 (inp=inp, out=io.out);
    | Empty =>
      spawn_fthread$ buffer (inp=io.inp, out=io.out);
    endmatch;
  }
  aux a io.inp;
}
```

Note that the empty check in `aux` can only succeed if the initial list `a` is empty because the singleton case in the second branch does not recurse.

4.1 Relation to option monad

Consider a sequence of partial functions:

```
pf:A -> B
pg:B -> C
ph:C -> D
```

The reverse functional composition chain

```
pf ⊙ pg ⊙ ph
```

which is spelled:

```
pf \odot pg \odot ph
```

is also a partial function. Since functional programming can't handle partial functions, we can lift these partial functions to

```
tf:A -> opt[B]
tg:B -> opt[C]
th:C -> opt[D]
```

where the function returns `None` if the underlying partial function is undefined, or `Some v` if it is, and the result would be `v`. For example the partial function `p_recip`

```
fun p_recip (d:double) => 1.0 / d;
```

can be lifted to the total function

```
fun t_recip (d:double) =>
  if d == 0.0 then None[double]
  else Some (1.0 / d)
;
```

Unfortunately, such totalised function cannot now be combined by ordinary composition because the codomain of one is no longer the domain of the next. To solve this problem, we need a new kind of composition, and I will show now one way to solve this problem.

Felix has monads:

```
class Monad [M: TYPE->TYPE] {
  virtual fun bind[a,b]: M a * (a -> M b) -> M b;
  virtual fun ret[a]: a -> M a;
}
```

It is conventional to use the infix operator `>>=` for `bind`. We can define the option monad:

```
typedef fun opt_f (T:TYPE):TYPE=>opt[T];

instance Monad[opt_f]
{
  fun ret[T] (x:T)=>Some x;
  fun bind[U,V] (init:opt[U], f:U->opt[V]) =>
    match init with
    | None => None[V]
    | Some u => f u
  ;
}
```

Now, we can use the monad to compose the functions:

```
var r : opt[D] = ret a >>= tf >>= tg >>= th;
```

Here, the `ret` is a lift of a plain value to the monad type. But there is another way:

```
var r = None[D];
run (a.value |-> tf.filter |-> tg.filter |-> th.filter
  |-> Some.function |-> store &r
);
```

using coroutines. Recall the `value` chip is a oneshot lift equivalent to the monadic `ret`, the pipeline operator is equivalent to monadic `bind`, and we tailed the pipeline with a conversion to an option type and a store.

The `filter` chip converts the option returning function to a transducer which does nothing on failure, but writes the result out on success, in other words, it takes the total function and converts it back to a partial function.

It would be natural to represent the original partial functions directly as coroutines: there is no need to go through the totalising lift, only to throw it away again with the `filter` chip. For example:

```
chip co_recip
  connector io
    pin inp: %<double
    pin out: %>double
{
  var d = read inp;
  if x != 0.0 perform write (io.out, 1/d);
}
```

This chip is a transducer, but it is not a functional transducer because it does not write an output for every input. I showed a oneshot but you can add a loop to make it continuous.

The pipeline operator will combine chips which directly implement partial functions, in the sense that they write the result out if there is one and do nothing if there is not. Unlike functions, chips can represent partial functions naturally. We don't need the totalisation or the option monad `bind` combinator.

Monads are sometimes considered to be way to do imperative programming in a functional language, that is, to sequence operations.

On the other hand, with coroutines, pipelines and channels are semantically defined to sequence events because coroutine semantics are entirely about event ordering. What we have shown here is that, at least for modelling partial functions, you do not actually need either option type or monads. Pipelines are intrinsically monadic.

Chapter 5

Acyclic Flow

Acyclic 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, f);
  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, 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
```

This looks complicated, but look at the diagran shown in 5.1. There are some tricks in the code: the `x` and `y` sources are reused, which is only safe because they're constant sources, and the `add`, `sub`, and `div` chips are one shots.

Exercise (Hard). If we wanted to make this system accept a list of pairs and process them, printing the values of x and y and the quotient, or an error message, what would we need to do?

5.1 Multi Writer

The following chip is a vital component with an interesting structure:

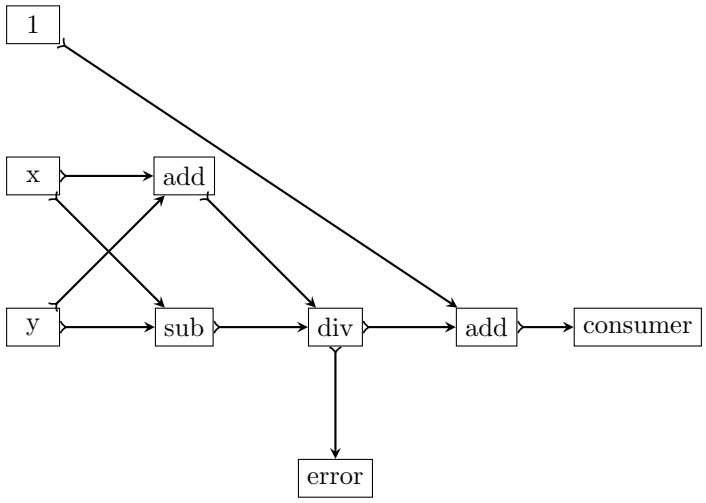

Figure 5.1: Flow in simple formula

```
chip tryall_list[D,C with Str[D]] (a: list[iochip_t[D,C]])
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    for h in a do
      var lin,lout = mk_ioschannel_pair[D]();
      spawn_fthread (h (inp=lin, out=io.out));
      write (lout,x);
    done
  done
}
```

Clearly, it repeatedly reads a value from its input channel, and writes the value down all the output channels in the supplied list. Getting that work as specified was not easy!

You may think, you could just read a value and write down all the channels. But this does not work because one of the channels may block the write, and then the whole chip is frozen up! So we have to spawn a new fibre for each value and each channel so that if the channel is blocked, the other writes can proceed.

What this means, however, is that, because the channel is reachable from the chip, whilst any one channel is not blocked, attempts to write will be stacked up for every input on every blocked channel.

5.2 Encoding and Decoding Unions

We will now examine two fundamental devices that correspond to construction and decoding of algebraic sum types. We will consider an example union:

```
union U = A of int | B of string;
```

In a functional setting values of type U are specified by using the injection functions

```
A: int -> U
B: string -> U
```

and decoded with a pattern match:

```
var a = A 42;
var b = B "Hello";
proc p(u:U) {
  match u with
  | A i => println$ "A argument is " + i.str;
  | B s => println$ "B argument is " + s;
  endmatch;
}
p a;
p b;
```

Our aim is to show the analogue of these operations using coroutines.

5.2.1 Encoding

Example

Encoding a union is easy:

```
chip encodeU
  connector io
    pin inpA: %<int
    pin inpB: %<string
    pin outU: %>U
{
   device wrapA = A.function;
   device wrapB = B.function;
   circuit
     wire io.inpA to wrapA.inp
     wire io.inpB to wrapB.inp
     wire io.outU to wrapA.out
     wire io.outU to wrapB.out
  endciruict
}
```

The `wrapA` device is a transducer that reads an int `i` and writes the U value `A i`, the `wrapB` device is a transducer that reads a string `s` and writes the U value `B s`. These transducers both write their U value to the same output pin. Each value fed to either of the encoder input pins is written out wrapped as a U to the output pin.

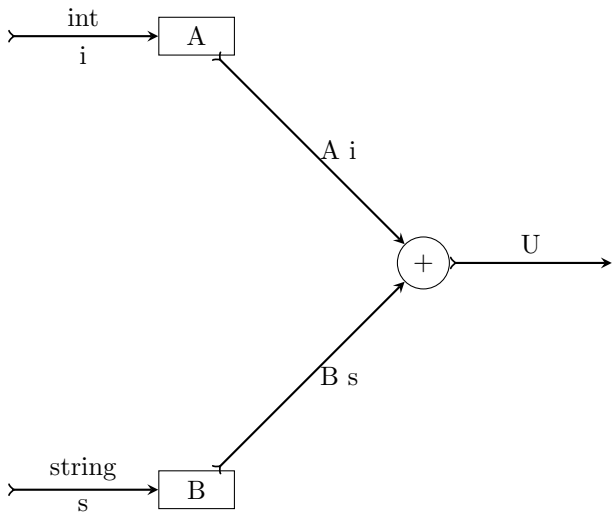

Figure 5.2: Encoder

Generic Encoder `_encode[U]`

Felix provides a generic union encoder `_encode[U]` which requires a union type `U` as an argument and produces an encoder with N input pins labelled with

the constructor names, and an output pin labelled with the union base name. Note that if the union is polymorphic it must be instantiated, possibly with an existential type variable.

If a union type has a constant constructor, it will be treated as if it had a unit argument.

5.2.2 Decoding

Example

To decode the stream of U's created we reverse the procedure:

```
chip decodeU
  connector io
    pin outA: %>int
    pin outB: %>string
    pin inpU: %<U
{
  while true do
    var u = read io.inpU;
    match u with
    | A i => write (io.outA, i);
    | B s => write (io.outB, s);
    endmatch;
  done
}
```

Here, we read a U, decode it with a pattern match, and write the argument out down one of two channels, depending on which constructor was used.

It's easy to generalise this to an arbitrary union. There is only one trick, which is that the `read` generator cannot be used for a unit value, so the read procedure has to be used instead, for example:

```
    var u: unit;
    read (channel, &u);
```

to prevent the read being discarded by the optimiser. The variable u above will actually be discarded: there's only one value of type unit so there's no need to store it in a variable.

A more difficult issue arises from constant constructors. For example in:

```
union opt[T] = None | Some of T;
```

Here, `None` appears to have a void argument. In fact, it is not a constructor, rather the constructor is:

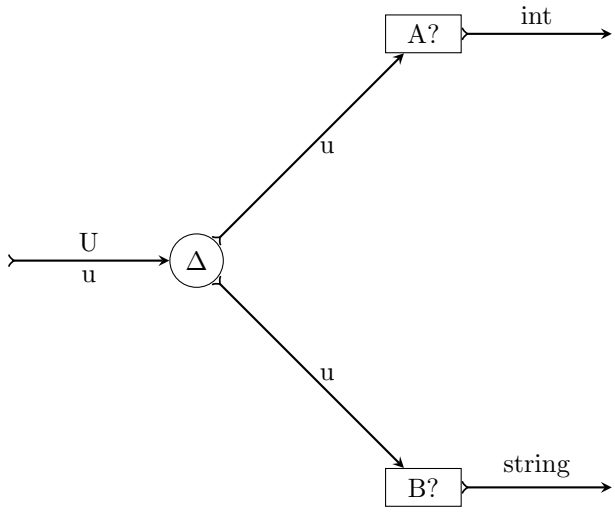

Figure 5.3: Decoder

Alternate implementation with duplicator and selective filters.

```
    None'[T] : unit -> opt[T]
    val None[T] = None'[T] ();
```

In other words, so called constant constructors actually name the result of applying a constructor function which accepts a unit to a unit value. They can do this because the result of applying a pure function to a unit is necessarily invariant.

So it is sensible, given a constant constructor, for the encoder to write a unit value, and the decoder to try to read one. However there is also an interpretation where we would write a void instead, which means, not actually writing anything, the reader, expecting nothing, simply skips any attempt to read. In this interpretation, the constants are lost from the streams, indeed the corresponding channels can be removed.

The former interpretation is closer to the functional model, the latter is a purer interpretation. The pure interpretation doesn't make a lot of sense unless chips are *clocked* so that the missing events are reintroduced by synchronised timing. We will see a bit later than in translating a list into a stream and vice versa both interpretations are useful.

Generic Decoder `_decode[U]`

Felix provides a generic union decoder `_decode[U]` which requires a union type `U` as an argument and produces an decoder with N output pins labelled with the constructor names, and an input pin labelled with the union base name. Note that if the union is polymorphic it must be instantiated, possibly with an existential type variable.

If a union type has a constant constructor, it will be treated as if it had a unit argument.

5.3 Symmetry of Encoders and Decoders

Encoders and Decoders for a given union type U are inverses.

Encoding a union from multiple channels and connecting the output to the corresponding decoder produces pins supplying the same output data as a set of buffers, one for each channel.

Similarly, decoding a union from a single channel and connecting its outputs to the corresponding input pins of an encoder produces the same effect as a buffer of the union type.

5.4 Decoding and Encoding Product Types

In order to fully decode a union we also need to encode and decode tuples and records: cartesian products. Products are dual to sums, they are characterised by projection functions:

```
typedef P = int * string;
pi0: P -> int;
pi1: P -> string;
var p: P = (42,"Hello");

typedef R = (A:int, B: string);
piA: R -> int
piB: B -> string
var r: R = (A=42, B="Hello");
```

We will deal with tuples first. To construct a tuple we need both the A and B values. We will read them in order in this example from two distinct channels. This is enough if the sources are independent. In a purely functional setting this is guarranteed since the argument expressions may not have side effects.

```
chip encodeP
  connector io
    pin i0: %<int
    pin i1: %<string
    pin P: %>int * string
{
  while true do
    var a = read io.i0;
    var b = read io.i1;
    write (io.P, (a,b));
  done
}
```

A record constructor is similar. To decode a tuple means to extract both its values:

```
chip decodeP
  connector io
    pin o0: %>int
    pin o1: %>string
    pin P: %<int * string
{
  while true do
    var a,b = read io.P;
    write (io.o0, a);
    write (io.o1, b);
  done
}
```

and again, the record case is similar. We can also split the projections:

```
chip decode0
  connector io
    pin o0: %>int
    pin P: %<int * string
{
  while true do
    var a,b = read io.P;
    write (io.o0, a);
  done
}

chip decode1
  connector io
    pin o0: %>int
    pin P: %<int * string
{
  while true do
    var a,b = read io.P;
    write (io.o1, b);
  done
}
```

5.5 Duality of products and sums

THIS BIT IS TOO WAFFLY.

We can also define our original decoder using a duplicator and two projections:

```
chip dup[T]
  connector io
    pin inp: %<T
    pin out1: %>T
    pin out2: %>T
{
  while true do
    var x = read inp;
    write (io.out1, x);
    write (io.out2, x);;
  done
}
```

and now we just connect the two projection filters to the two channels. The `dup` chip is also known as the diagonal operator and written in greek as Δ. In imperative programming, it is called `copy` or `clone`.

There is a dual chip named `merge` which reads two channels of the same type,

and outputs the result on a single channel, the mathematical name is ∇ which pronounced Nabla, is an upside down delta, it is squashs a sum type all of whose constructors take the same argument by removing the constructors.

In order to understand more deeply the categorical relationship between products and sums, we will consider the notion of a heterogenous array containing strings and integers. Of course the array has to be homogenous so we do this:

```
union U = A of int | B of string;
var a : U ^ 3 = A 1, B "Hello", A 42;
```

Now we can index the array with values of the sum type 3, which is the sum of three units, and the result is a U, which can then be decoded. However it is possible to directly index a heterogenous array:

```
var a : int * string * int = 1, "Hello", 42;
var v0 : int = a.0;
var v1 : string = a.1;
var v2 : int = a.2;
```

but the results have different types. We can fix this by unifying them:

```
var a : int * string * int = 1, "Hello", 42;
fun idx (a: int * string * int) (i:int) : U =>
  match i with
  | 0 => A a.0
  | 1 => B a.1
  | 2 => A a.2
;
for i in 0..<3 do
  var x : U = idx (a,i);
  ..
done
```

More generally, any tuple can be dynamically indexed by using a generalised projection which returns the dual of the tuple type:

```
pi: int * string * int -> 3 -> int + string + int
```

where the RHS is an anonymous sum with numbered constructors. In Felix the notation is ugly, the results of the projection in order would be:

```
typedef S = int + string + int;
case 0 of S (1)
case 1 of S ("Hello")
case 2 of S (42)
```

We can pattern match these anonymous sums:

```
    match s with
    | case 0 i => println$ "Case 0, int=" + i.str;
    | case 1 s => println$ "Case 1, int=" + s;
    | case 2 i => println$ "Case 2, int=" + i.str;
```

Now the point is that if all the sums have the same argument type, we can squash the constructor away, and that is precisely what happens with ordinary array indexing. In particular, this is a special case because for an arbitrary tuple you can't squash the constructor. The key is that the core indexing operation applies to any product and returns a value of the dual sum type.

The important thing here is that the fields of a product and constructors of a sum are perfectly dualised and indexable, the special case of arrays is handled using N-ary versions of Δ and ∇ to introduce and eliminate indexes.

The categorical relations here are crucial for understanding the translations of these operations to the temporal domain. We begin to picture that the decoder chip is a real sum, which is used to lift the functional sum type into a temporal one. The functional sum is a cheat, the real sum is the temporal one.

With products, we actually have a drop instead of a lift. The functional product is a spatial one, and the temporal version of it feels unnatural. The reason is that what we really want for a temporal product is a sequence of data going down a channel, as for a list source.

In fact, you can write any values you want down channels, they do not have to be the same type. The result is not ill-typed, provided the reader expects the values to have the correct type.

In the temporal domain we do not need to wrap the values in constructors to unify them, we just need to check that the type sequence the reader and writer use agree. We don't have a type checker that can do that at the moment!

Chapter 6

Cyclic Flow

6.1 Feedback

Circuits with feedback create a cyclic flow. Lets try an experiment.

```
chip fibit
  connector io
    pin inp1: %<int
    pin inp2: %<int
    pin out1: %>int
    pin out2: %>int
    pin result: %>int
{
  while true do
  var x1 = read io.inp1;
  var x2 = read io.inp2;
  var x3 = x1 + x2;
  write (io.out1, x2);
  write (io.out2, x3);
  write (io.result, x3);
  done

}

device v1 = value 1;
device v2 = value 1;
device printer = procedure println[int];
circuit
  connect v1.out, fibit.inp1
  connect v2.out, fibit.inp2
  connect fibit.out1, fibit.inp1
  connect fibit.out2, fibit.inp2
  connect fibit.result, printer
endciruit
```

This is an attempt to run the fibonacci series in which starts with the values 1, then 1, thereafter each value is the sum of the two previous ones.

We start with two oneshot values to get the circuit running. We read them, the older one first, then the newer one, add them up, then write the newer one to the older pin, and the sum to the newer pin. We also write the sum to the result pin.

But .. nothing happens! The problem is that the chip is writing to itself, and that never works, because the write can't return until after a matching read is done, but the read can't happen because the device is suspended in the write.

The solution is to add buffers.

```
device v1 = value 1;
device v2 = value 1;
device b1 = buffer;
device b2 = buffer;
device printer = procedure println[int];
circuit
  connect v1.out, fibit.inp1
  connect v2.out, fibit.inp2
  connect fibit.out1, b1.inp // out1->inp1 via buffer
  connect b1.out, fibit.inp1
  connect fibit.out2, b2.inp // out2->inp2 via buffer
  connect b2.out, fibit.inp2
  connect fibit.result, printer
endciruit
```

There is another way to do this, using a spatial store instead of a temporal one:

```
chip fibit  (var inp1:int, var inp2:int)
  connector io
    pin result: %>int
{
  while true do
    var x1 = inp1;
    var x2 = inp2;
    var x3 = x1 + x2;
    inp1 = x2;
    inp2 = x3;
    write (io.result, x3);
  done
}
```

We just used variables instead of buffers. They're storage locations, which are of course ways to hold values over time, which is of course what a buffer is. Indeed the buffer chip does exactly that, it reads the value into a variable then writes it. Effectively we just optimised the calculation!

Chapter 7

Base Recognisers

7.1 Recognisers

A *recogniser* for a language $L \subset \Sigma^*$ is a machine which can decide if a string $s \in \Sigma^*$ is an element of L; in other words determine the truth of the predicate $s \in L$.

In this chapter we introduce a data structure used for scanning strings, some specific primitive recognisers, and some general purpose chips which can be used as recogniser combinators.

The idea of a base recogniser is that you have a string, and want to see if it matches a some simple specification. We then form the ability to recognise more complex specifications by providing ways to combine these base recognisers called *combinators*. The idea is that combinators are simple to use, the combinations derived have easily predictable behaviour, and finally that the combined components have good, predictable, executable performance.

7.2 Reasoning Principles

Decoupling and recoupling components allows a complex problem to be broken down into local reasoning about the behaviour of the base components and the combinators, so that together with standard rules of inference we can deduce the behaviour of combinations. We should note there are two aspects of this: reading and writing. Reading is analytical reasoning: given a combination, try to calculate what it does. Writing is synthetic reasoning: given a problem, try to calculate a combination which solves the problem.

A well designed system must support both inspection and construction, since

software development is an interactive cycle requiring alternate and simultaneous application of both skills. Conventional regular expressions, for example, are easy to construct but impossible to read and so fail this test: regular definitions are clearly a superior technology.

Our system will use basic coroutines as our base unit of computation, and will use circuit constructors as the combinators used to combine them.

7.3 Abstraction

What we do is pass to the recogniser a pointer into the string. The recogniser chips reads the pointer and examines the string from that point to see if some of the string after that pointer matches what the chip is supposed to recognise, and if so, it writes out a pointer to the first character in the string which does not match.

But here is where you have to make the big paradigm shift! If the chip doesn't recognise anything at the location the pointer indicates, it doesn't write anything!

This is quite different behaviour from a function! A function would return an option type, saying `Some p` if it recognised something and returning a pointer `p` to the first unrecognised character, or `None` if it didn't recognise anything. Functions always have to output something! If they didn't, the function would be non-terminating, and that means an infinite loop would cause the whole program to fail.

Coroutines, on the other hand, regularly report failure by keeping quiet! They don't know what to do, so they do nothing.

The abstraction here allows us to understand not only how to design base recognisers but also what the results of applying combinators to them should be.

7.4 Buffers

Now, here is the kind of pointer to the string we will use, for some reason its called a `Buffer`.

```
struct Buffer
{
  sp: varray[char];
  pos: int;

  fun atend => self.pos >= self.sp.len.int;

  fun get =>
    if self.atend then char ""
    else (self.sp) . (self.pos)
  ;

  proc next {
    if not self*.atend do
      pre_incr self.pos;
    done
  }

  fun advanced =>
    if self.atend then self
    else Buffer (self.sp, self.pos + 1)
  ;

  fun lookahead (i:int) =>
    if self.pos + i > self.sp.len.int then char ""
    elif self.pos + i < 0 then char ""
    else (self.sp) . (self.pos + i)
  ;

  fun stl_end => Buffer (self.sp,self.sp.len.int);
}
```

Figure 7.1: lst:Type Buffer (lib Recog)

Two instance variables make a `Buffer`, a `varray` of `char` called `sp`, for the string and an `int` named `pos` for as an index into the string to mark the current position. The array is represented by a pointer and its contents will not be changed during processing.

The `atend` method detects if the current position is just after the last character. The `get` method gets the current character as a string, or returns the empty string at the end.

The `next` method is mutator that increments the current position if it is not at the end, wherease the `advanced` method returns a copy of the object advanced one, or at the same position if it is at end, this method is purely functional.

Finally the `lookahead` method returns the character `i` positions ahead of the current position, if there is one, or an empty string if that position is past the end.

We also provide some constructors:

```
ctor Buffer (p:varray[char]) =>
  Buffer (p,0)
;

// excludes a trailing nul byte!
ctor Buffer (p:string) =>
  Buffer (p.varray_nonul,0)
;

ctor Buffer (p: &string) =>
  Buffer (*p)
;
```

a conversion to a printable string representation:

```
instance Str[Buffer] {
  fun str (b:Buffer) => "@"+b.pos.str;
}
```

and an equality and total ordering operator which ignores the underlying string and just compares the positions: these operators only work correctly for the same underlying array.

```
instance Eq[Buffer] {
  fun == (a:Buffer, b:Buffer) => a.pos == b.pos;
}
instance Tord[Buffer] {
  fun < (a:Buffer, b:Buffer) => a.pos < b.pos;
}
```

It isn't really correct but it will do for our purposes, if the base strings are distinct the buffers are aren't really comparable.

Finally a simularly hacked extractor finds the string between two positions, including the first position and excluding the second:

```
ctor string (a:Buffer, b:Buffer) =
{
  var x = "";
  for i in a.pos ..< b.pos do
    x += a.sp.i;
  done
  return x;
}
```

Now we have a the data type which will be transmitted along channels of the recogniser circuit, we can define it by

```
typedef recog_t = BaseChips::iochip_t[Buffer,Buffer];
```

which says that a `recog_t` is just a chip which takes an input `Buffer` value, tries to recognise a part of the underlying string starting at the input position, and writes out the position just past the end of the recognised substring, if one exists.

Note that if we don't recognise anything, nothing is written, the chip just cycles around and tries again with another input. This behaviour is not evident from the data type, it would be approximated in the control type if we have a system for control types, but we don't. On failure, coroutines typically either do nothing or write on a special error channel allowing a normal or alternate continuation to take further action.

In our parser we specially want failures to have no consequences! Unlike functional code, failure doesn't return an error code to the caller, requiring the caller to check it. Rather, it simply fails to propagate and translate information, allowing the algorithm to continue exploring successful paths and allowing failure paths to simply die off without ceremony. You don't need an exception to abort a computation.

In principle, this is like a non-returning function, however since computations are built from many interconnected *active* fibres, non-completion is not only not undesirable, it is the *prefered method* of constructing algorithms!

7.5 Primitives

We now present some basic primitive recognisers.

7.5.1 String matcher

This chip just matches its argument as a string.

```
chip match_string (s:string)
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
nextmatch:>
  var b = read io.inp;
  for i in 0..< s.len.int do
    if s.[i] != b.get goto nextmatch;
    b&.next;
  done
  write (io.out, b);
  goto nextmatch;
}
```

The code reads a string position represented by a `Buffer`, and tries to match the argument string. If it does, it writes out the position one past the end of the matched substring, otherwise it does nothing. In either case it then goes back and reads another position and tries again.

7.5.2 Whitespace matcher

This transducer matches a sequence of whitespace characters, defined as any character less that or equal to a space character. It always succeeds, that is, if there is no white space character at the current position, it just returns the current position. That is, it is a *total function*.

```
chip match_white
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
  while true do
    var b = read io.inp;
    while not b.atend and b.get <= char ' '
      perform b&.next
    ;
    write (io.out,b);
  done
}
```

7.5.3 C++ comment matcher

Matches two / characters in succession, followed by any sequence of characters, up to and including the first newline character, or, up to the end of the string.

This matcher also cannot fail, if it cannot match a C++ comment it emits the input position. It is a total function.

```
chip match_cxx_comment
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
again:>
  var b = read io.inp;
  var b_saved = b;

  if b.get != char "/" goto bad;
  b&.next;

  if b.get != char "/" goto bad;
  b&.next;

  while not b.atend and not (b.get == char "\n")
     perform b&.next
  ;
  b&.next; // works fine even if atend
ok:>
  write (io.out,b);
  goto again;
bad:>
  write (io.out,b_saved);
  goto again;
}
```

7.5.4 Nested C comment matcher

This is another total function that matches C style comments, except that it also recognises nested comments. On the other hand it is a bit naive in that it also finds nested delimiters in enclosed C++ comments and strings.

```
chip match_nested_c_comment
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
again:>
  var depth = 0;
  var b = read io.inp;
  var b_saved = b;
  if b.get != char "/" goto bad;
  b&.next;
  if b.get != char "*" goto bad;

nest:>
  b&.next;
  ++depth;

scan:>
  if b.get == "/" do // start nested comment
    b&.next;
    if b.get == "*" goto nest;
    goto scan;
  done

  if b.get == "*" do // end comment group
    b&.next;
    if b.get == "/" goto unnest;
    goto scan;
  done

  b&.next;
  goto scan;

unnest:>
  b&.next;
  --depth;
  if depth > 0 goto scan;
  write (io.out,b);
  goto again;

bad:>
  write (io.out,b_saved);
  goto again;
}
```

7.5.5 Regular Expression matcher

This is a generally useful matcher that uses Felix system library binding of Google's RE2 package for matching.

```
chip match_regex (r:RE2)
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
  while true do
    var b = read io.inp;
    var matched = varray[StringPiece] (1uz,StringPiece());
    var result = Match
      (
        r,StringPiece(b.sp),b.pos,
        ANCHOR_START,matched.stl_begin,1
      )
    ;
    if result do
      var b2 = Buffer (b.sp,b.pos+matched.0.len.int);
      write(io.out,b2);
    done
  done
}

device cident_matcher =
  match_regex (RE2 "[A-Za-z][A-Za-z0-9_]*")
;
device decimal_integer_matcher =
  match_regex (RE2 "[0-9]+")
;
device decimal_float_matcher =
  match_regex (RE2 "[0-9]+\\.[0-9]+")
;
```

We also included a matcher for C identifiers which uses the regexp matching chip, and simple decimal integer matcher.

7.5.6 End of String matcher

A particularly interesting chip! It reads a position and writes it out if, and only if, its at the end of the string.

By adding this chip to the end of a pipline you can force the whole string to be matched by the rest of the pipeline, instead of just a prefix.

```
chip eos_matcher
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
  while true do
    var x = read io.inp;
    if x.atend perform write (io,out,x);
  done
}
```

7.6 Combinators

7.6.1 Delegators

We present two versions of delegators, or proxies, which are chips that delegate processing to another chip addressed by a pointer passed as an argument. We will need delegation for forward referencing and recursion, when we need the action of a chip that hasn't been defined yet.

The following transducer accepts a pointer to another chip and delegates processing to the pointed at chip which is dereferenced each read, allowing the target to be changed during processing:

```
chip deref_each_read[D,C] (p:&iochip_t[D,C])
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    var rinp,rout = mk_ioschannel_pair[D]();
    spawn_fthread ((*p) (inp=rinp, out=io.out));
    // println£ "Deref_each_read: write " + io.out.address.str;
    write (rout,x);
  done
}
```

Whereas this transducer also accepts a pointer to another chip and delegates processing to the pointed at chip, but it only dereferences the pointer once, after the first data is read.

```
chip deref_first_read[D,C] (p:&iochip_t[D,C])
  connector io
    pin inp: %<D
    pin out: %>C
{
  var x = read io.inp;
  var rinp,rout = mk_ioschannel_pair[D]();
  spawn_fthread ((*p) (inp=rinp, out=io.out));
  write (rout,x);
  while true do
    x = read io.inp;
    write (rout,x);
  done
}
```

7.7 Alternatives

This chip is just a copy buffer with the name changed to epsilon, to suggest it recognises everything.

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

7.7.1 Optional

The optional chip uses `tryall_list` to both pass some data through unmodified, and also to pass it through modified by an arbitrary transducer. Models regular expression A?

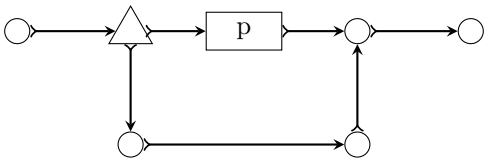

Figure 7.2: optional

```
chip optional[T] (p:iochip_t[T,T])
  connector io
    pin inp: %<T
    pin out: %>T
{
  device both = tryall_list ([
    p,
    epsilon[T]
  ]);
  circuit
    wire io.inp to both.inp
    wire io.out to both.out
  endcircuit
}
```

7.7.2 One or More

The `oneormore_match` matches the regular expression `A+`. It requires at least one match.

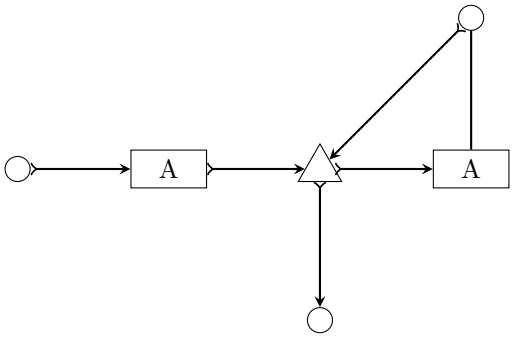

Figure 7.3: oneormore

```
chip oneormore_matcher[T] (A:iochip_t[T,T])
connector chans
  pin inp: %<T
  pin out: %>T
{
 device As = oneormore_matcher A;
 device As2 = pipeline_list (A,As).list;
 device Ass = tryall_list (A, As2).list;
 circuit
   wire chans.inp to Ass.inp
   wire chans.out to Ass.out
 endcircuit
}
```

And of course, this one matches `A*`.

7.7.3 Zero or More

```
chip zeroormore_matcher[T] (A:iochip_t[T,T])
connector chans
  pin inp: %<T
  pin out: %>T
{
 device As = oneormore_matcher A;
 device Ass = tryall_list (epsilon[T], As).list;
 circuit
   wire chans.inp to Ass.inp
   wire chans.out to Ass.out
 endcircuit
}
```

7.7.4 Handling Ambiguity

In the **?? ??** we provided a matcher for integers and a simplified floating point format without an exponent part. Now consider matching the string:

```
"42.78"
```

Lets build a ciruit which tries to detect if the prefix of a string matches a number.

```
chip number
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
  device num = tryall_list
    ([
      decimal_integer_matcher,
      decimal_float_matcher
    ])
  ;
  circuit
     wire io.inp to num.inp
     wire io.out to num.out
  endcircuit
}
```

The problem is that the input produces two outputs: the whole string is a float but the first two characters are also an integer.

What can we do? One common specification for recognisers is that only the longest match should be used. You should note that this strategy will eliminate possible parses prematurely; that is, by excluding what is currently ambiguous, later elimination of the longest match will lead to failure, while keeping both matches might succeed.

Nevertheless, longest match may be required, so the question becomes how to implement it. If we were using functional code we would get a set with all the matches, and just select the longest one. But we're using coroutines not functions, do we can't do that.

Right? Well .. actually we can!

```
chip longest_match (a: list[recog_t])
  connector io
    pin inp: %<Buffer
    pin out: %>Buffer
{
  var x = read io.inp;
  var results = None[Buffer];
  proc storemax[T with Tord [T]] (p: &opt[T]) (a:T) {
    match *p with
    | None => p <- Some a;
    | Some v => if a > v perform p <- Some a;
    endmatch;
  }
  for r in a call
    run (x.value |-> r |-> (storemax &results).procedure)
  ;
  match results with
  | None => ;
  | Some answer => write (io.out, answer);
  endmatch;
}
```

The trick is that the `run` operation on a pipeline terminated by a drop will capture all the results, as if the code were functional. Indeed, `run` is an ordinary subroutine!

What we have done here, in terms of exploring a tree of parses, is commonly known as a *cut*, in which we prematurely prune whole subtrees based on some predictive heuristic. In this case, it's not an optimisation but a requirement of the specification.

In general, a cuts are a technique for finding the spatial form of a finite temporal sequence. The `run` subroutine is the key because it returns control when the fibre scheduler it has invoked no longer has any work to perform, in other words, when it can no longer reach any active fibres. Active fibres may still exist on the scheduler that called it, and it could create fibres itself which are suspended on channels it called knows about.

As this would be a side-effect were the run nested in a function, the programmer must take care in that case to capture all such effects. Generators are allowed to have side effects, so it may be useful to use a generator instead. However, generators are really only intended to have internal mutable state which should not escape the generator abstraction. In particular generators have the same type as functions, so generator closures might accidentally be passed to a function which expected a functional, not generative, argument.

In our example, our code is safe, and indeed the `longest` chip is pure. The reasoning is that the `processor` pipeline only reads data from the channel with

endpoints `wi,wo` which is local to the chip, and the drop only occurs to `results` which is also local. So no effects escape from the `longest` chip: what it writes is purely a function of what it reads, no other behaviour can be detected by a circuit that uses it.

The other side of purity reasoning is cleanup reasoning. We need to know that, should this chip become unreachable, then anything it creates also becomes unreachable. This means no references to anything it creates can escape the chip. The local variable `results` is addressed, which is dangerous, but the address is passed only to the `drop` chip which itself becomes unreachable at the end of the `run` operation, except for its inclusion in the `process` pipeline which is a local variable which is not addressed, and therefore is only locally reachable.

Reachability reasoning is vital in functional programs too so this is not a new burden on programmers, however the reasoning is complicated by the need to consider fibres and channels as potentially reachable objects.

It's vital to understand that the `cicruit` statement does something more than just relieve the programmer of mundane channel creation and spawning tasks. Critically, it hides the names of the channels and in making the secret *prevents* the programmer from inadvertantly reaching them, thereby ensuring that the blocked or starved circuit components will be reaped by the garbage collector. In other words, it provides a kind of abstraction through anonymity which helps keep components pure.

7.8 Recogniser test harness

We need to test our primitives, and later, combinations of them. What would a test harness look like?

The first thing is to write a simple function.

```
fun tr (r: recog_t) (s:string) : bool =
{
  var result = false;
  run (
    source (Buffer s)
    |-> oneshot[Buffer]
    |-> r
    |-> eos_matcher
    |-> function (fun (x:Buffer) => true)
    |-> store &result
  );
  return result;
}
```

The pipline starts with a source emitting the initial Buffer repeatedly which is

cut down by the oneshot to a single output. Then we run the recogniser. After that we run a check that we have reached the end of the string. Then we have a chip that reads the resulting end position and ignores it, writing true instead, which the store sink puts in the result variable.

So if the match goes through, and matches the whole string, the function `tr` returns true. If the pipeline dies, `run` returns control without the initial value of false being modified, and so `tr` returns false.

This shows a lift using the `source` chip, and a drop using the `store` chip, terminated by the `run` procedure, which returns when there are no active fibres left. You will note, the drop is in two parts: the data drop is done by the `store` chip, the control drop is done by verbof the drop, data and control, are not automatically synchronised.

7.8.1 A paradigm shift

One of the key themes in this work is that fibration extends conventional functional and imperative programming styles. The new technology does not replace the old methods, but should be used in conjunction with them. Understanding coroutines requires a paradigm shift, but this does not mean abandoning the old ways, just recognising they do not form a complete development methodology. So too, fibration is not the last word, but it adds something vital missing to the old techniques: symmetry.

Technically, the dual of a function is a cofunction, which is basically an iterator. General coroutines, having multiple channels, are dual to OO style objects: we have multiple synchronisation vehicles instead of multiple methods, we have a common thread of control instead of an object shared between methods. OO isolates the data using functional abstraction, so access is limited to the use of the fixed set of methods. Coroutines isolate their state using channels instead.

Objects are passive slaves providing methods as services to the client of the object, supporing observers that report state and mutators that change it. Objects are reactive.

Coroutines are active threads of control. They also isolate the client from their state, but fibres actively cooperate with their environment as peers. Instead of calling a method, and the method responding, fibres accept requests and send data and the system behavior is determined by circuit connectivity.

Both object methods and coroutines need to obey invariants, however the object invariants are imposed on their spatial state, whereas the coroutines invariants are temporal: they reflect ordering rules.

When something is broken, objects panic: typically by throwing an exception or terminating the program. Fibres, on the other hand, simply ignore faults. So bad OO programs blow up, whereas bad circuits simply die.

Chapter 8

Grammars

8.1 Theory

Given a set of symbols Σ the set of all finite sequences of these symbols is denoted by Σ^*, which is called the *Kleene closure* of the set. A *language* on Σ is any subset of Σ^*. The elements of Σ are called *terminal symbols*. The strings of a language are called *sentences* in the language.

A *production grammar* is a machine which can generate languages. We start by introducing a set of symbols N disjoint from Σ, called *nonterminal symbols*, and defining $S = \Sigma \cup N$, the set of symbols. A *production sequence* P is a sequence of symbols, and a *grammar library* G is a mapping

$$G : S \to 2^{\mathbf{P}}$$

where $\mathbf{P}$ is the set of all possible production sequences on S, so that $2^{\mathbf{P}}$ is the power set, or set of all subsets of $\mathbf{P}$. A grammar is a grammar library and a selected nonterminal s which is called the *start symbol*. A *production* is the association of a nonterminal symbol n and one of the production sequences p in the set of production sequences $G(p)$.

In other words, a grammar associates each nonterminal symbol with a set of productions, and selects a particular nonterminal as special.

We can use the grammar to generate a language as follows: beginning with the start symbol, select an associated production and write it down. This is known as a *sential form*. Now, pick any nonterminal symbol in the sential form, and replace it with one of the productions with which it is associated. This is known as a *production step* or *derivation step.*

We say the sentential form produced is *directly derived* from the original one. You can repeat this procedure a number of times, any sentential form so obtained is said to be *derived* from the any previous one in the sequence of expansions.

You may eventually end up with a sentential form containing no nonterminals, such a form is then specified to be a sentence of the language generated by the grammar. We can then define the language generated by a grammar as the set of all sentences which can be derived from the start symbol.

There are some things to note here. Our definition of a grammar is slightly unconventional in that we start with a grammar library and pair it with a start symbol. This means that some of the library entries will be useless because, begining with the start symbol, no sentential form we derive can possibly contain that nonterminal. Such a nonterminal is said to be *unreachable*. The set of all nonterminals which can appear in some sential form after some sequence of derivations is performed is known as the *closure* of the start symbol, the other nonterminals are said to be *garbage*.

There may also be productions which simply associate a nonterminal with the singlton sequence consisting of the nonterminal itself. If used in a derivation step, the resulting sentential form is unchanged. Such a production is said to be *useless* since it achieves no progress in deriving a sentence. A nonterminal X might also be associated with singleton production Y which in turn is associated again with X, so that whilst replacing X once with Y derives a new sentential form, replacing that Y again with X leaves us back where we started. Such a deriviation sequence is said to be *cyclic*. We will come back to this issue later because the solutions for plain and labelled grammars are different!

Two grammars may produce the same language, in which case they are said to be *equivalent*.

If, presented with a sentential form, one always chooses to replace the left most nonterminal, the resulting sequence of derivations is called a *leftmost derivation*. Chosing the rightmost, unsurprisingly, is a *rightmost derivation*. One can also choose replacements for all the nonterminals and replace them simultaneously, or, at least, in a fixed order such as left to right, this is a *breadth first* derivation. Two derivation sequences may be considered equivalent if the production replacing each particular nonterminal in a sentential form is the same, irrespective of the order of the actual substitution. The equivalence class of such derivation sequences exhibits a structure common to all of them, and different from any other, which has the hierachical shape and is known as a *derivation tree*.

You can picture a derivation tree by considering a sentential form and under each nonterminal writing the production sequence replacing it, with an arrow from the nonterminal to its associated sequence. Begin with the start symbol and apply the rule recursively until there are no more nonterminals to replace.

8.1.1 Labelled Grammars

We will now introduce a small modification to the notion of a grammar called a *labelled grammar*. This is the same as a grammar except a name is attached

to each production. As usually the names will all be distinct from each other, and distinct from any terminal or nonterminal symbol.

The label is useful, when generating strings, to annotate the resulting sentence with pairs of markers representing the production which was chosen: you can put a start marker indexed by the label and an end marker indexed by the label at the start and end of each production which is included in the generated sentential form, but ignored when considing the final sentence generated.

Equivalently, in the production tree, the label can be attached to each node node.

If we closely examine a labelled grammar a suprising fact emerges. Consider

```
S -> L
L -> I L
L -> E
E -> unit
I -> int
```

You may not recognise this yet but let me rewrite it as a labeled grammar:

```
S -> List: L
L -> Cons: I L
L -> Empty: E
E -> Unit: unit
I -> Integer: int
```

and now again, with different syntax:

```
union S = List of L;
union L =
  | Cons of I * L
  | Empty of E
;
union I = Integer of int;
union E = Unit of unit;
```

What this basically shows is that we should consider a nonterminal associated with a type of the same name, and the labels as type constructors. The production for each constructor is just the type of the constructor argument. In this model, each terminal is a primitive type.

It is not hard to see now that every inductive type corresponds to a (labelled) grammar, and every labelled grammar to a type. Of course, the labelled derivation tree is then just a value of the type.

Given any grammar, it is possible to refactor the grammar to create another, equivalent grammar. We may want, for example, to remove the possibility of cyclic derivations. Unfortunately, our ability to do this and allow a recogniser for the language to produce the desired result, it is much more difficult if we wish

to produce a derivation tree because we are no longer free to add productions or nonterminals, since if used, unrecognised nodes in the tree would be created.

We will present a solution to this problem later, by upgrading again our notion of a labelled grammar to an *action grammar* specifically designed so that it may be refactored but still allow a parser to produce the expected derivation tree. Roughly speaking an action grammar is a labelled grammar equiped with additional codes which allow refactoring steps to be invertible, so that the parser can parse the modified grammar, but the tree building process translates the tree from the modified to original form in the process of building it. Unlike other parsing machinery, the tree does not need to be translated after being built. The critical refactoring step which is supported is left recursion elimination, however cycle removal is possible as well.

8.2 Partially Labelled Grammars

We will briefly introduce another extension, the partially labelled grammar. Roughly, sometimes a grammar production has common sequences we would like to factor out. In a grammar, we can simply introduce a new nonterminal associated with the common subsequence, and replace occurences of the subsequence with the new nonterminal. Note, the new nonterminal supports exactly one alternative.

Unfortunately we cannot do this in a labelled grammar, because we would have to introduce a new type and type constructor, which would change the derivation tree. To get around this we add yet another kind of symbol, disjoint from the others, called a *macro symbol*, which is used like a nonterminal. Exactly one production must be associated with the macro, and it is not given a constructor label.

8.3 Representation

We will now take a break from coroutines and do some functional programming, because this is a fairly easy way to do the job we have to do in preparation for the next section.

What we need is a representation of a grammar, because later in this work we're going to show how to make a parser with coroutines. Just to keep you on your toes, we're going to use a very powerful functional programming technique known as *open recursion*.

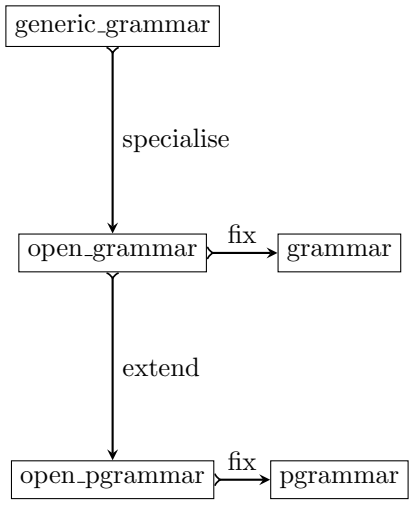

Figure 8.1: Grammar Development

8.4 Generic Grammar Datatype

The first thing we need is a data structure representing a grammar:

```
typedef generic_gramentry_t[T] = string * T;
typedef generic_gramlib_t[T] = list[generic_gramentry_t[T]];
typedef generic_grammar_t[T] = string * generic_gramlib_t[T];
```

What this says is that a grammar entry is a symbol, represented by a string, which is used as an identifier, and some other information we're not specifying.

A grammar library is a set of grammar entries. We're using a list because its easy. We need to take care the list entries are unique, so it represents a set, and similarly that we do not inadventently depend on the ordering the list exhibits.

A grammar is a pair consisting of a grammar library and a selected symbol called the *start symbol*.

Now, a grammar entry is intended to be a nonterminal and a production. But we made the product a value of an unspecified type, `T` so we could fill in the details later. You will be surprised how we do that! For now know that the secret of open recursion is to first use a parameter to define a structure, and then later set the parameter to that structure, so that initially the structure is flat but parametrised, and the recursion is introduced by setting the parameter to itself, establishing a cycle.

8.5 Generic Closure Algorithm

Now we're going to show you how to use this idea to find the closure of the grammar. This is a list of all the symbols reachable from the start symbol.

The algorithm is a stock standard closure operation. Given some symbol r, we find all the symbols $s1, s2, \ldots$ it knows about directly. We then add them to the set of known symbols, then pick a symbol from the set we haven't examined before, and find out what it knows about. We continue until we have examine all the symbols, so that all the symbols they know about are already in the set.

My algorithm splits the set in two pieces: ones we have already examines and ones we have not. Each new symbol is added to the set of symbols we do not know about. However we ignore a symbol we have already seen, even if it isn't examined yet, since it is already in one of the two sets.

Now you might ask .. indeed you should be asking .. how can we find the symbols associated directly with a symbol r if the associated information is of an unknown type `T`?

And the answer is: we can't! But the caller has to know how to find all the symbols in a `T` so we just require them to pass in a function that can do it!

First, we have a recursive induction step:

```
fun generic_cls[T]
  (generic_add: list[string] -> T -> list[string])
  (lib:generic_gramlib_t[T])
  (unprocessed: list[string])
  (processed:list[string])
: list[string]
=>
  match unprocessed with
  | Empty => processed
  | Cons (h,tail) =>
    if h in processed then
      generic_cls generic_add lib tail processed
    else
      match find lib h with
      | Some p =>
        let unprocessed = generic_add tail p in
        let p = Cons (h, processed) in
        generic_cls generic_add lib unprocessed p
      | None =>
        fun_fail[list[string]] ("MISSING NONTERMINAL " + h)
      endmatch
  endmatch
;
```

Now we complete the induction with an initial case:

```
fun generic_closure[T]
  (generic_add: list[string] -> T -> list[string])
  (g:generic_grammar_t[T])
: list[string] =>
  match g with
  | start, lib =>
    generic_cls generic_add lib ([start]) Empty[string]
;
```

You may notice this algorithm doesn't do quite what I said it would: we actually add all the associated symbols to the unprocessed set, even if some of them are already processed. The reason is that the check for this is done before looking for the associated symbols, anyhow, so its equivalent to not needing the check and because we didn't add the symbol to the unprocessed set. This simplifies the requirements on the `generic_add` function.

8.6 Open Grammar

Ok, so now, we define the *concrete* type of a production we desire in two stages. We use open recursion! First we define the flat, non-recursive, *open* form:

```
union open_prod_t[T] =
| Terminal of string * recog_t
| Nonterminal of string
| Epsilon
| Seq of list[T]
| Alt of list[T]
;
```

```
typedef open_gramentry_t[T] = string * open_prod_t[T];
typedef open_gramlib_t[T] = list[open_gramentry_t[T]];
typedef open_grammar_t[T] = string * open_gramlib_t[T];
```

This function takes a grammar library and a new entry, and returns a new grammar library with the entry added.

```
fun open_add_prod[T]
  (aux: list[string] -> T -> list[string])
  (acc:list[string]) (p: open_prod_t[T])
: list[string] =>
  match p with
  | Terminal _ => acc
  | Nonterminal name => Cons (name, acc)
  | Epsilon => acc
  | Seq ps => fold_left aux acc ps
  | Alt ps => fold_left aux acc ps
  endmatch
;
```

And here's the obligatory pretty printer:

Pretty Printer

```
instance[T with Str[T]] Str[open_prod_t[T]]
{
  fun str: open_prod_t[T] -> string =
  | Terminal (s,r) => '"' + s + '"'
  | Nonterminal name => name
  | Epsilon => "Eps"
  | Seq ss => "(" + catmap " " (str of T) ss + ")"
  | Alt ss => "[" + catmap " | " (str of T) ss + "]"
  ;
}
```

8.7 Closed Grammar

Now we tie the recursive knot with a circular definition which effects a fixpoint operation:

```
typedef gramentry_t = open_gramentry_t[prod_t];
typedef gramlib_t = open_gramlib_t[prod_t];
typedef grammar_t = open_grammar_t[prod_t];
```

It takes a while to grok this! You may wonder why we didn't just make the data type recursive in the first place. The answer is profound and subtle: the flat structure we chose is easy to extend by adding new cases, we can then fixate the extended type. This leaves the original fixation unaffected, but it makes the extended type recursive. In particular, the `Seq` and `Alt` cases would allow lists of the extended type, not just the original type, in other words, the extension is *covariant* meaning the extension propagates to the case arguments too. This

would not be so if we extended the original recursive type, because the recursive knot has already been tied.

Construction

```
fun add_prod(acc:list[string]) (p:prod_t) : list[string] =>
  fix open_add_prod[prod_t] acc p
;
```

Reified Closure

```
fun closure (g:grammar_t): list[string] =>
  generic_closure[prod_t] add_prod g
;
```

8.8 Open/Closed Principle

Open recursion is a technique which obeys a vital law of modularity known as the *open/closed principle*, originally espoused by Bertrand Meyer in Object Oriented Software Construction. The principle states a module must be simultaneously closed so it can be used, and open for extension. What is most interesting is that classes in Object Orientation, proposed to support this principle, where extension is by inheritance, actually fail it, and precisely if a method of an base object with an argument of the based type, that is, a binary operator, is overridden in a derived class by a method with an argument of the derived type, then the type system is unsound: the argument must in fact be contravariant, but the requirement is for covariance. This is known as the covariance problem and it destroys, utterly and completely, the belief that object orientation is the basis of a sound software development paradigm.

8.9 Properties

8.9.1 Nullable property

A nonterminal is said to be *nullable* if it can produce the empty string. We can calculate this inductively by observing a nonterminal is nullable if one of its alternatives is the empty string ϵ or a nullable nonterminal. Note this algorithm assumes terminals can't be null!

In a conventional grammar, terminals are symbols, so it would be a category error to even ask if they could be null, since they're not strings by characters,

a completely different type. However we are constructing a scannerless parser, without tokenisation, and a terminal is any computable nonempty string.

Calculate if a production RHS is nullable, depends on knowing if a nonterminal is nullable.

```
fun nullable_prod
  (lib:gramlib_t)
  (e:prod_t)
  (trail:list[string])
=>
  match e with
  | Terminal _ => false
  | Seq es => fold_left (fun (acc:bool) (sym:prod_t) =>
      acc and (nullable_prod lib sym trail)) true es

  | Alt es => fold_left (fun (acc:bool) (sym:prod_t) =>
      acc or (nullable_prod lib sym trail)) false es

  | Nonterminal nt => nullable_nt lib nt trail
  | Epsilon => true
;
```

Calculate if a nonterminal is nullable, which depends on knowing if a production is nullable.

```
fun nullable_nt
  (lib: gramlib_t)
  (nt:string)
  (trail:list[string])
: bool =>
  if nt in trail then false else
  match find lib nt with
  | None => false
  | Some e => nullable_prod lib e (nt ! trail)
;
```

Close the inductions with a base case.

```
fun is_nullable_prod (lib:gramlib_t) (e:prod_t) =>
  nullable_prod lib e Empty[string]
;

fun is_nullable_nt (lib:gramlib_t) (nt:string) =>
  nullable_nt lib nt Empty[string]
;
```

8.9.2 Recursive Property

```
fun recursive_prod
  (lib:gramlib_t)
  (e:prod_t)
  (orig:string)
  (trail:list[string])
=>
  match e with
  | Terminal _ => false
  | Seq es =>
    fold_left (fun (acc:bool) (sym:prod_t) =>
      acc or (recursive_prod lib sym orig trail)) false es

  | Alt es =>
    fold_left (fun (acc:bool) (sym:prod_t) =>
      acc or (recursive_prod lib sym orig trail)) false es

  | Nonterminal nt =>
    if nt == orig then true
    else recursive_nt lib nt orig trail

  | Epsilon => false
;
```

A nonterminal is recursive if it can produce a sentential form containing itself.

```
fun recursive_nt
  (lib: gramlib_t)
  (nt:string)
  (orig:string)
  (trail:list[string])
: bool =>
  if nt in trail then false else
  match find lib nt with
  | None => false
  | Some e => recursive_prod lib e orig (nt ! trail)
;


fun is_recursive_nt (lib:gramlib_t) (nt:string) =>
  recursive_nt lib nt nt Empty[string]
;
```

8.9.3 Left Recursive Property

A nonterminal is left recursive if it can produce a sentential form in which the first symbol is itself.

```
fun left_recursive_prod
  (lib:gramlib_t)
  (e:prod_t)
  (orig:string)
  (trail:list[string])
=>
  match e with
  | Terminal _ => false

  | Seq es =>
    let fun aux (es:list[prod_t]) =>
      match es with
      | Empty => false
      | Cons (head, tail) =>
        if left_recursive_prod lib head orig trail then true
        elif is_nullable_prod lib head then aux tail
        else false
      endmatch
    in
    aux es

  | Alt es => fold_left (fun (acc:bool) (sym:prod_t) =>
      acc or (left_recursive_prod lib sym orig trail)) false es

  | Nonterminal nt =>
    if nt == orig then true
    else left_recursive_nt lib nt orig trail

  | Epsilon => false
;
```

```
fun left_recursive_nt
  (lib: gramlib_t)
  (nt:string)
  (orig:string)
  (trail:list[string])
: bool =>
  if nt in trail then false else
  match find lib nt with
  | None => false
  | Some e => left_recursive_prod lib e orig (nt ! trail)
;

fun is_left_recursive_nt (
  lib:gramlib_t)
  (nt:string)
=>
  left_recursive_nt lib nt nt Empty[string]
;
```

8.10 Normal Forms

Our grammar uses a weakened EBNF form in which we allow productions to contain expressions using the `Seq` and `Alt` constructors. However a conventional grammar requires production RHS to only consist of a sequence of symbols, where a symbol is either a terminal or nonterminal. Optionally, some specifications also allow ϵ as the RHS of a nonterminal other than the start symbol.

The following routine unpacks a production library into a normal form in which all `Seq` and `Alt` combinators have been removed at the cost of introducing some fresh nonterminals with synthesised names.

We do this by replacing each `Seq` expression with a fresh nonterminal, and defining it to be the contents of the constructor. An `Alt` expression is also replaced by a fresh nonterminal, which is defined by a series of productions of that nonterminal, one RHS for each alternative.

The process is repeated until no `Seq` or `Alt` terms are left, which leaves only epsilons, terminals, and nonterminals in the library. The library is then said to be *flat* because all the nesting is eliminated.

```
fun unpack
  (fresh:1->string)
  (head:string, p:prod_t)
: gramlib_t =
{
 var out = Empty[gramentry_t];
 match p with
 | Epsilon => out = ([head,p]);
 | Terminal _ => out = ([head,Seq ([p])]);
 | Nonterminal s => out= ([head,Seq ([p])]);

 | Seq ps =>
   var newseq = Empty[prod_t];
   for term in ps do
     match term with
     | Epsilon => ;
     | Nonterminal _ => newseq = term ! newseq;
     | Terminal _ => newseq = term ! newseq;
     | _ =>
       var newhead = fresh();
       newseq = Nonterminal[prod_t] newhead ! newseq;
       out = unpack fresh (newhead,term);
     endmatch;
   done

   match newseq with
   | Empty => out = (head,Epsilon[prod_t]) ! out;
   | _ => out = (head,Seq[prod_t] (rev newseq)) ! out;
   endmatch;

 | Alt ps =>
   iter
     (proc (p:prod_t) {
       out = unpack fresh (head,p) + out;
     })
     ps
   ;
 endmatch;
 return out;
}
```

```
fun normalise_lib (fresh:1->string) (lib:gramlib_t) = {
  var normalised = Empty[gramentry_t];
  for p in lib perform
    normalised = unpack fresh p + normalised;
  return normalised;
}
```

8.10.1 Another normal form

The normal form produced above allows multiple alternatives for a nonterminal to be scattered throughout the library, making it hard to find a complete definition.

To fix this we provide another normal form in which each symbol is defined exactly once. In principle, the RHS of each definition will be a `Alt` term, however, we replace an empty `Alt` with ϵ and a singleton with only one alternative with that alternative.

If the input to our routine is normalised by our first normalisation procedure, then the output has the property that each of these alternatives will be a nonempty list of symbols.

The result is then unique up to the order of the alternatives.

```
fun sort_merge (g:gramlib_t) : gramlib_t =>
 let
   fun enlt
     (a:gramentry_t, b:gramentry_t)
   : bool =>
     a.0 < b.0
 in
 merge (sort enlt g)
;
```

```
fun merge (var p:gramlib_t): gramlib_t =
{
  if p.len == 0uz return p;

  var out: gramlib_t;

  var key: string;
  var alts = Empty[prod_t];
  var cur: gramentry_t;

  proc fetch() {
    match p with
    | Cons (head,tail) => cur = head; p = tail;
    | Empty => assert false;
    endmatch;
  }

  proc dohead() { key = cur.0; alts = Empty[prod_t]; }
  proc dofoot() { out = (key,Alt alts) ! out;  }
  proc dobreak() { dofoot; dohead; }
  proc check() { if key != cur.0 call dobreak; }

  fetch;
  dohead;
  while p.len > 0uz do
    check;
    alts = cur.1 ! alts;
    fetch;
  done
  check;
  alts = cur.1 ! alts;
  dofoot;
  return out;
}
```

Chapter 9

Recogniser from Grammar

9.1 Building a Recogniser

We are now read for the big job: to build a recogniser from a grammar.

9.1.1 Helper

First we need a helper which finds the index of a nonterminal definition in an array of definition:

```
fun find (v:varray[ntdef_t]) (nt:string) : size =
{
  for i in 0uz ..< v.len do
    if v.i.0 == nt return i;
  done
  assert false;
}
```

9.1.2 Primary Renderer

Now the core function, which converts a production expression into a recogniser. Because a function cannot use chip combinators or fibration primitives, the function uses lazy evaluation: it returns a procedure which when executed will do the appropriate operations.

```
fun render_prod
  (lib:gramlib_t,v:varray[ntdef_t])
  (p:prod_t)
: recog_t =>
  match p with
  | Terminal (s,r) => r

  | Epsilon => epsilon[Buffer]

  | Seq ps => pipeline_list ([
      map (fun (p:prod_t) => (render_prod (lib,v) p)) ps])

  | Alt ps => tryall_list ([
      map (fun (p:prod_t) => (render_prod (lib,v) p)) ps])

  | Nonterminal nt =>
    let idx = find v nt in
    let pslot = -(v.stl_begin + idx) in
    let pchip = pslot . 1 in
    deref_first_read pchip
  endmatch
;
```

Here is a detailed explanation.

Terminal

Since a terminal is defined with a debugging string s and a recogniser r, we just return the recogniser in that case.

Epsilon

Just returns `epsilon` which recognises a null string.

Seq

If we have a sequence, we return a pipeline of the chips corresponding to the entries in the sequence: we make the chips by recursively mapping the render function over the entries in the list.

Alt

If we have alternatives, we do the same mapping but combine the chips using the `tryall_list` combinator instead of a pipeline.

Nonterminal

The hard case is the nonterminal. We cannot just lookup the nonterminal and substitute its definition, because grammars can be recursive.

Every problem can be solved by adding yet another level of indirection! So what we do if find the index position of the definition of the nonterminal, calculate the address in the output arrayu v that where the definition will eventually be put, and then use the magical `deref_first_read` chip which will delegate its actions to the chip at the end of the pointer, but will only do so on demand, after its first data is read.

The varray access is messy. The `stl_begin` method returns an incrementable pointer to which we add the index `idx` we previously found using the `find` helper. We must then downgrade the resulting incrementable pointer to a non-incrementable pointer to which we can add the offset of the second component. The downgrade conversion is represented by the prefix `-` sign, it is statically, but not dynamically safe. The component selection is done by applying a pointer-to-tuple projection which returns a pointer to one of the tuple components. Components of tuples are numbered from 0 up, zero origin, so the component with index 1 is actually the second component. Due to a shortcut, we can just use a plain integer literal to designate the projection. Finally we can pass the pointer to the second component to the `deref_first_read` chip. The slot we have pointed at may not contain the correct chip at this time, however it should be filled in by the complete rendering process before it completes. This is why the dereference is delayed until the first data is read by the chip, since that is after the rendering is finished.

9.1.3 The generator chip

Since we have to combine coroutines inside a coroutine, not a function, we will make a chip which does the job.

This chip reads a whole grammar, and writes the corresponding recogniser.

First we find the transitive closure of the nonterminals reachable from the designated start symbol, since our data is a library of nonterminal definitions, not all the components in the library may be required for the specified start symbol.

We make an array to store the chips for each nonterminal in the closure and initialise them to the `epsilon` chip to expand the `varray` to the right size, and to assign the names of the nonterminals into the array.

Now we run the `render_prod` function for each nonterminal, which we then store in the array.

Then we just write the chip associated with the start symbol, that's our recogniser. Note that the varray storing the nonterminals and their recogniser is not lost, since it is reachable from the starting chip.

```
fun recogniser
  (start:string, lib:gramlib_t) : recog_t =
{
  var cl = closure (start,lib);

  // allocate a varray with a slot for each nonterminal
  var n = cl.len;
  var v = varray[string * recog_t] n;

  // populate the varray with the terminal names and a dummy chip
  for nt in cl call // initialise array
    push_back (v,(nt,BaseChips::epsilon[Buffer]))
  ;

  // now assign the real recogniser_base to the array
  var index = 0uz;
  for nt in cl do
    match find lib nt with
    | None => assert false;
    | Some prod =>
      // get wrapped recogniser
      var entry = render_prod (lib, v) prod;

      // address of the slot
      var pentry : &recog_t = (-(v.stl_begin+index)).1;

      // overwrite dummy value
      pentry <- entry;
    endmatch;
    ++index;
  done
  return v.(find v start).1;
}
```

And here's a simple function that checks if a string `s` is a member of the language specified by a grammar:

```
fun in (s:string) (g:grammar_t) =
{
  chip false_if_got (pr: &bool)
     connector io
       pin inp: %<Buffer
  {
    C_hack::ignore$ read io.inp;
    pr <- true;
  }
  var r = recogniers g;
  var result = false;
  run (s.value |-> r |-> eos_matcher |-> false_if_got &result);
  return result;
}
```

9.2 Testing

Chapter 10

Parsing

We are now ready to build our parser. We are going to this in a difficult way, leveraging our recogniser as the base. You can think of this as inheriting this technology, but the method of doing so is by using *open recursion*. The OO way doesn't work because it is not covariant, the functional technique allows us to retain covariance.

10.1 Action Grammars

But first, we have to extend our grammars to be *action grammars*. An action grammar is like a grammar, except that there are three types of symbols. As well as terminals and nonterminals, we also have a new kind of symbol called an *action*.

The idea of an action is simple enough. When the parser hits an action, it does it. Our parser will operate by a variant of recursive descent, but it wil actually be an LR parser!

In terms of parsing, actions are equivalent to *epsilon*, that is, they do not consume any input. Instead, they modify an entity called the parser stack which maintains the state of the parser. The modification will be purely functional and monadic, and this is absolutely essential.

I will show you the action data type now, but we will have to defer a complete understanding what it is for until later.

```
union action_t =
| Reduce of string * int
| Scroll of int
| Unscroll of int
| Pack of int
| Unpack
| Drop of int
| Swap
| Sequence of list[action_t]
;
```

As a rough guide, when the parsing is going along parsing a sequence of symbols and it hits a reduce action, the top n values on the stack are removed and replaced by a single list of those value labelled by the given string. This is known as a *reduce action*.

Normally when the parser runs, when it parses text corresponding to a symbol it has to push the recognised lexem onto the stack, this is known as a *shift action*. Parsers using this method are called *shift/reduce parsers*.

Our action machine has an auxilliary stack, and the scroll and unscroll actions roll value off the top of the parser stack onto the auxilliary stack, and back again (respectively). These operations are vital because our parser, perhaps uniquely, has a property which is unusual but essential: our parser allows you to refactor the grammar, parse the refactored grammar, and end up with exactly the same output, as if it had not been refactored.

Refactoring is absolutely essential because recursive descent parsers cannot cope with left recursive grammar productions. If you refactor the grammar to remove the left recursion, you would normally get a different, unwanted, parse tree than the one the original grammar would produce.

We solve this problem by *backtracking the parser stack* to the state it would have had earlier, performing the reduce action that would have occured then, and then reinstating the head of the stack. That's what the scroll and unscroll on the auxilliary stack are for.

The pack and unpack operations are helpers to assist use contructing parse tree nodes.

In fact, no one actually wants to produce a parse tree. Everyone wants to produce an abstracted version of the parse tree in which things like punctuation are removed. This is called an *abstract syntax tree* or AST.

Swap and Drop are the usual stack operations. You can see the action codes are actually instructions for a specialised stack machine like Forth.

Pretty printer

```
instance Str[action_t] {
  fun str: action_t -> string =
  | Reduce (s,n) => "Reduce(" + s + ","+n.str+")"
  | Scroll n => "Scroll " + n.str
  | Unscroll n => "Unscroll " + n.str
  | Pack n => "Pack " + n.str
  | Drop n => "Drop " + n.str
  | Swap => "Swap"
  | Sequence aa =>
    "Seq(" + catmap "," (str of action_t) aa + ")"
  ;
}
```

10.2 Action Grammar Extension

Now we have the actions, we extend our open grammar to include them:

```
union open_pgram_t[T] =
  | Action of action_t
  | Recog of open_prod_t[T]
;
```

Notice very carefully, we have extended the open version of the grammar, the one with a parameter. Our extension is to the flat version of the grammar, not the one with the recursive knot already tied by fixation.

```
instance[T with Str[T]] Str[open_pgram_t[T]]
{
  fun str: open_pgram_t[T] -> string =
  | Action a => "{" + a.str + "}"
  | Recog r => r.str
  ;
}
```

Now finally you can see how we reap the reward or reusability from open recursion! We use the pretty printer we defined for the actions in the first branch, but in the second branch we can delegate the pretty printing to the existing code for pretty printing recognisers.

All of our code above will continue to work unmodified, even if we extend the definition of a recogniser, and thanks to our type system, it will also work with any recogniser because in a production, we required the client to tag the recogniser with a label, for the purpose of saying what it does.

Now we define the new grammar type by fixating the open form:

```
typedef pgram_t = open_pgram_t[pgram_t];
```

I have to repeat what we did: we started with an open form of the grammar and then defined a closed form by fixation.

Then we extended the open grammar adding new terms to it, and then close that form with fixation.

Our type system does have polymorphic variants, which would have allowed adding the new action term directly to the end of the original type, but we have chosen to use the more conventional union constructor and a wrapper layer, because although it is more verbose, the nominal typing is easier to understand and and the compiler will give more definite error messages.

In any case we now repeat the same kinds of extensions we used before for grammar entries, grammar libraries and grammars:

```
typedef open_pgramentry_t[T] = string * open_pgram_t[T];
typedef open_pgramlib_t[T] = list[open_pgramentry_t[T]];
typedef open_pgrammar_t[T] = string * open_pgramlib_t[T];
```

and now we close them:

```
typedef pgramentry_t = open_pgramentry_t[pgram_t];
typedef pgramlib_t = open_pgramlib_t[pgram_t];
typedef pgrammar_t = open_pgrammar_t[pgram_t];

instance Str[pgramlib_t] {
  fun str (lib: pgramlib_t) : string =
  {
    var s = "";
    match nt,ex in lib do
      s += nt + ":\n";
      s += "  " + ex.str+"\n";
    done
    return s;
  }
}
```

Notice we didn't have to use the fixpoint operator here: we just plug in the already recursive type `pgram_t`.

10.3 Parser stack

Now here is the promised parser stack. First we have to define the type of a lexeme, which is span of characters in the string we're parsing.

```
typedef lexeme = (start:Buffer, finish:Buffer);
```

Now the type of a value on the stack with the usual pretty printer:

```
union stack_node_t =
| RTerminal of string * lexeme
| RNonterminal of string * list[stack_node_t]
;

instance Str[stack_node_t] {
  fun str: stack_node_t -> string =
  | RTerminal (s,x) => s+"("+string (x.start,x.finish)+")"
  | RNonterminal (s,xs) =>
    s + "(" + catmap "," (str of stack_node_t) xs + ")"
  ;
}
```

Here an `RTerminal` is the result of parsing a terminal, and consists of the name of a recogniser that recognised a string, together with the lexeme that it recognised. We use a lexeme, not just a string, so we know where in the input we are up to.

Our stack entry `RNonterminal` is the result of parsing a nonterminal. It consists of a string, which will be the name of the nonterminal, and a list of stack nodes. The idea is that the `Reduce` action will pull off the nodes that the nonterminal production put on there, and label them with the nonterminal name. In fact, since a nonterminal can have multiple productions, what we actually want is an AST, so what we put there is an invented named called a *constructor*. We may also fiddle with the list of values on the stack, for example, removing punctuation, before constructing a list of essential information for the constructor. That fiddling is done by the other action instructions, thats what they're for. You can use `Drop`, for example, to throw away punctuation.

So here's the type of a parser stack with the usual pretty printer:

```
typedef parser_stack_t = list[stack_node_t];

instance Str[parser_stack_t] {
  fun str (x:parser_stack_t) =>
    catmap "; " (str of stack_node_t) x
  ;
}
```

At any point, we will have "code" starting in some parser state and trying to proceed to a new state, the state is just the current string position together with a parser stack. Oh, and of course a pretty printer is required:

```
typedef parser_state_t =
(
  pos: Buffer,
  stack: parser_stack_t
);

instance Str[parser_state_t] {
  fun str (x:parser_state_t) =>
    x.pos.str + ":  " + x.stack.str
  ;
}
```

10.4 Actions

Now we're ready to define our parser:

```
typedef parser_t = BaseChips::iochip_t[parser_state_t,parser_state_t];
```

You might have thought a parser is a function, which returns a parse tree or an AST. However, this is not what a parsing function would do. The parsing might fail, and, if the grammar is ambiguous there could be more than one parse tree. So actually, a functional parser, in general, must return a set of parse trees, not just one. The empty set means there were no parses.

With fibration, the equivalent model uses a transducer, which attempts to parse the string from the given input position with multiple pipelines, one for each alternative. If the pipeline succeeds it writes the end position recognised, otherwise it does nothing. The transducer loops for the next input position.

So at the output we see a stream of matches, the non-matching pathways producing nothing. The output correspond precisely to the set of matches a functional parser would return, they're just spread out over time instead of space.

The first thing we need is a chip that does a shift action. It runs a recogniser, and then, if the recogniser succeeds, it pushes the corresponding lexeme onto the parser stack with the supplied label.

```
chip ActionShift (label:string) (r: recog_t)
  connector io
    pin inp: %<parser_state_t
    pin out: %>parser_state_t
{
  // We need to use a secondary chip so that if the recogniser
  // writes no output, this chip will block on it and die
  // without killing off the ActionShift chip.
  chip handler
    connector inner
      pin inp: %<parser_state_t
  {
    var inp = read inner.inp;

    var ri,wi = #mk_ioschannel_pair[Buffer];
    var ro,wo = #mk_ioschannel_pair[Buffer];
    circuit
      wire ri to r.inp
      wire wo to r.out
    endcircuit

    var ipos = inp.pos;
    write (wi, ipos);
    var opos = read ro;
    var entry = RTerminal (label, (start = ipos, finish = opos));
    //println£ "ActionShift " + label + " write " + io.out.address.str;
    write (io.out, (pos = opos, stack = entry ! inp.stack));
  }

  while true do
    var inp = read io.inp;
    var ri,wi = #mk_ioschannel_pair[parser_state_t];
   circuit
      wire wi to handler.inp
    endcircuit
    write (wi, inp);
  done
}
```

Now, a special variant of the shift chip recognises two strings, but it only pushes the lexeme of the second one.

You may ask why and the answer is fairly simple: the idea is just that the first recogniser is for whitespace, which we skip over first, the second one is the actual word we want to recognise. This allows us to construct a grammar which makes no provision for whitespace, and then throw in whitespace before each symbol systematically and automatically.

```
chip ActionSecond (label:string) (r1: recog_t) (r2: recog_t)
  connector io
    pin inp: %<parser_state_t
    pin out: %>parser_state_t
{

  chip handler
    connector inner
      pin inp: %<parser_state_t
  {
    var inp = read inner.inp;

    var ri1,wi1 = #mk_ioschannel_pair[Buffer];
    var ro1,wo1 = #mk_ioschannel_pair[Buffer];
    var ri2,wi2 = #mk_ioschannel_pair[Buffer];
    var ro2,wo2 = #mk_ioschannel_pair[Buffer];
    circuit
      wire ri1 to r1.inp
      wire wo1 to r1.out
      wire ri2 to r2.inp
      wire wo2 to r2.out
    endcircuit

    // whitespace
    var pos1 = inp.pos;
    write (wi1, pos1);
    var pos2 = read ro1;

    // terminal
    write (wi2, pos2);
    var pos3 = read ro2;

    var entry = RTerminal (label, (start = pos2, finish = pos3));
    //println£ "ActionSecond " + label + " write " + io.out.address.str;
    write (io.out, (pos = pos3, stack = entry ! inp.stack));
  }

  while true do
    var inp = read io.inp;
    var ri,wi = #mk_ioschannel_pair[parser_state_t];
    circuit
      wire wi to handler.inp
    endcircuit
    write (wi, inp);
  done
}
```

Now we can define a function that does actions.

This function takes as input an auxilliary parser stack `aux`, the main parser stack `s`, and an action `a`, and returns a pair of the auxilliary parser stack and the new parser stack after the action is done. It is a pure function.

```
fun doaction (aux: parser_stack_t,s:parser_stack_t) (a:action_t) =>
  match a with
  | Reduce (label,n) =>
    let revhead,tail = revsplit n s in
    aux,RNonterminal (label,revhead) ! tail

  | Drop n => aux,drop n s

  | Swap => aux,
    match s with
    | e1 ! e2 ! tail => e2 ! e1 ! tail
    | _ => s
    endmatch

  | Scroll n => let s,a = scroll (s,aux) n in a,s
  | Unscroll n => scroll (aux,s) n

  | Pack n =>
    let revhead,tail = revsplit n s in
    aux,RNonterminal ("_Tuple",revhead) ! tail

  | Unpack =>
    match s with
    | RNonterminal (_,ss) ! tail => aux, ss + tail
    | _ => aux,s
    endmatch

  | Sequence actions =>
    fold_left (fun (aux:parser_stack_t,s:parser_stack_t) (a:action_t) =>
      doaction (aux,s) a)
      (aux,s)
      actions

  endmatch
;

fun doaction (s:parser_stack_t) (a:action_t) =>
  let _,s = doaction (Empty[stack_node_t], s) a in
  s
;
```

The second variant just uses an initially empty auxilliary stack. These stacks are only used to connect subactions of a parser action. The auxilliary stack never propagates outside master action.

Here is the action applied by a chip to a parser state (the previous function only applied to a stack).

```
chip ActionGeneral (a:action_t)
  connector io
    pin inp: %<parser_state_t
    pin out: %>parser_state_t
{
  while true do
    var i = read io.inp;
    var pos = i.pos;
    var stack = doaction i.stack a;
    //println£ "ActionGeneral ["+a.str+"] write " + io.out.address.str;
    write (io.out, (pos=pos, stack=stack));
  done
}
```

10.5 The parser

What we need to do now is build a parser from an action grammar. We proceed as we did with recognisers.

```
typedef pntdef_t = string * parser_t;

fun find (v:varray[pntdef_t]) (nt:string) : size =
{
  for i in 0uz ..< v.len do
    if v.i.0 == nt return i;
  done
  assert false;
}

fun render_pgram
  (lib:pgramlib_t,v:varray[pntdef_t])
  (white:recog_t)
  (p:pgram_t)
: parser_t =>
  match p with
  | Recog r =>
    match r with
    | Terminal (s,r) => ActionSecond s white r
    | Epsilon => BaseChips::epsilon[parser_state_t]
    | Seq ps => BaseChips::pipeline_list (
        map (fun (p:pgram_t) => (render_pgram (lib,v) white p)) ps)
    | Alt ps =>  BaseChips::tryall_list (
        map (fun (p:pgram_t) => (render_pgram (lib,v) white p)) ps)
    | Nonterminal nt =>
       let  idx : size = find v nt in
         let pslot : &pntdef_t = -(v.stl_begin + idx) in
         let pchip : &parser_t = pslot . 1 in
         BaseChips::deref_each_read pchip
    endmatch
  | Action a => ActionGeneral a
;
```

```
fun open_add_pgram[T]
  (aux: list[string] -> T -> list[string])
  (acc:list[string]) (p: open_pgram_t[T])
: list[string] =>
  match p with
  | Recog r => open_add_prod[T] aux acc r
  | Action a => acc
  endmatch
;

fun add_pgram (acc:list[string]) (p:pgram_t) : list[string] =>
  fix open_add_pgram[pgram_t] acc p
;

fun closure (g:pgrammar_t): list[string] =>
  generic_closure[pgram_t] add_pgram g
;
```

```
chip make_parser_from_grammar (white:recog_t)
  connector io
    pin inp: %<pgrammar_t
    pin out: %>parser_t
{

  while true do
    // read in the grammar
    var start, lib = read io.inp;

    // calculate the transitive closure of nonterminals
    // from the start symbol
    var cl = closure (start,lib);

    // allocate a varray with a slot for each nonterminal
    var n = cl.len;
    var v = varray[string * parser_t] n;

    // populate the varray with the terminal names and a dummy chip
    for nt in cl call // initialise array
      push_back (v,(nt,BaseChips::epsilon[parser_state_t]))
    ;

    // now assign the real recognisers to the array
    var index = 0uz;
    for nt in cl do
      match find lib nt with
      | None => assert false;
      | Some prod =>
        // get wrapped parser
        var xprod_closure = render_pgram (lib, v) white prod;

        // undo the wrappingt
        var entry = #xprod_closure;

        // address of the slot
        var pentry : &parser_t = (-(v.stl_begin+index)).1;

        // overwrite dummy value
        pentry <- entry;
      endmatch;
      ++index;
    done
    write (io.out, (v.(find v start).1));
  done
}
```

```
gen make_parser_from_grammar (g:pgrammar_t) (white:recog_t) : parser_t =
{
  var parsr: parser_t;
  var sched = #fibre_scheduler;
  spawn_fthread sched {
    var gri,gwi = mk_ioschannel_pair[pgrammar_t]();
    var gro,gwo = mk_ioschannel_pair[parser_t]();
    spawn_fthread (make_parser_from_grammar white (inp=gri,out=gwo));
    write (gwi, g);
    parsr = read gro;
  };
  sched.run;
  return parsr;
}
```

```
gen run_parser_on_string (parsr:parser_t) (s:string) : list[parser_state_t] =
{
  var results = Empty[parser_state_t];
  var b = Buffer s;
  var ps : parser_state_t = (pos=b, stack=Empty[stack_node_t]);
  var sched = #fibre_scheduler;
  spawn_fthread sched {
    var ri,wi = mk_ioschannel_pair[parser_state_t]();
    var ro,wo = mk_ioschannel_pair[parser_state_t]();
    spawn_fthread (parsr (inp=ri, out=wo));
    write (wi,ps);
    while true do
      var result = read ro;
      results = result ! results;
      //println£ "Test1: End pos (should be 14)=" + result.str;
    done
  };
  sched.run;
  return results;
}
```

Chapter 11

Grammar Refactoring

```
// replace internal sub-expressions with fresh nonterminals
fun unpack (fresh:1->string) (head:string, p:pgram_t) : pgramlib_t =
{
 var out = Empty[pgramentry_t];
 match p with
 | Action a => out = ([head,p]);
 | Recog Epsilon => out = ([head,p]);
 | Recog (Terminal _) => out = ([head,Recog (Seq[pgram_t] ([p]))]);
 | Recog (Nonterminal s) => out= ([head,Recog (Seq[pgram_t] ([p]))]);

 | Recog (Seq ps) =>
   var newseq = Empty[pgram_t];
   for term in ps do
     match term with
     | Action _ => newseq = term ! newseq;
     | Recog Epsilon => ;
     | Recog (Nonterminal _) => newseq = term ! newseq;
     | Recog (Terminal _) => newseq = term ! newseq;
     | _ =>
       var newhead = fresh();
       newseq = Recog (Nonterminal[pgram_t] newhead) ! newseq;
       out = unpack fresh (newhead,term);
     endmatch;
   done

   match newseq with
   | Empty => out = (head,Recog (Epsilon[pgram_t])) ! out;
   | _ => out = (head,Recog (Seq[pgram_t] (rev newseq))) ! out;
   endmatch;

 | Recog (Alt ps) =>
   iter (proc (p:pgram_t) { out = unpack fresh (head,p) + out; }) ps;
 endmatch;
 return out;
}
```

```
// expand internal sub-expressions, return a list of symbol sequences
// the outer list are the alternatives and the inner ones sequences
// IN REVERSE ORDER!
fun expand_aux (p:pgram_t) : list[list[pgram_t]] =
{
 var out = ([Empty[pgram_t]]);
 match p with
 // add symbol to each alternative
 | Recog Epsilon => ;
 | ( Action a
 | Recog (Terminal _)
 | Recog (Nonterminal s)) =>
   out = map (fun (ss: list[pgram_t]) => Cons (p,ss)) out;

 // A sequence is unpacked by successively unpacking each
 // symbol. The result is then prepended to each alternative.
 | Recog (Seq ps) =>
   for term in ps do
     var tmp = expand_aux term;
     var out2 = Empty[list[pgram_t]];
     for left in tmp perform
       for right in out perform
         out2 += left + right;
     out = out2;
   done

 | Recog (Alt ps) =>
   var alts = cat (map expand_aux ps);
   out2 = Empty[list[pgram_t]];
   for left in alts perform
     for right in out perform
       out2 += left + right;
   out = out2;

 endmatch;
 return out;
}

fun expand (p:pgram_t) : pgram_t =>
  let ps = expand_aux p in
  Recog (Alt[pgram_t] (map (fun (seqs: list[pgram_t]) => Recog (Seq[pgram_t] (rev seq
;
```

```
// in p replace nonterminal name with value (where q=name,value)
fun substitute (q:pgramentry_t) (p:pgram_t)=>
  let name,value = q in
  match p with
  | Recog (Nonterminal s) when name == s => value
  | Recog (Seq ls) => Recog (Seq[pgram_t] (map (substitute q) ls))
  | Recog (Alt ls) => Recog (Alt[pgram_t] (map (substitute q) ls))
  | _ => p
;
```

```
// direct left recursion eliminator
// assumes A = A alpha | beta form
// outputs
// A = beta A'
// A' = alpha A' | Eps
//
// BETTER
//
// A = beta | beta A'
// A' = alpha A' | alpha
//
// since this is Epsilon free

  fun direct_left_recursion_el\imination
   (fresh:1->string)
   (lib:pgramlib_t)
  =
  {
   var outgram = Empty[pgramentry_t];
   for ntdef in lib do
     var nt,expr = ntdef;
     var alphas = Empty[list[pgram_t]];
     var betas = Empty[list[pgram_t]];
  // where does Epsilon go??
     match expr with
     | Recog (Alt alts) =>
       for alt in alts do
         match alt with
         | Recog (Seq (Cons (Recog (Nonterminal $(nt)),tail))) => alphas = tail ! alpha
         | Recog (Seq b) => betas = b ! betas;
         | x => betas = ([x]) ! betas;

         //| x => println£ "EDLR, unexpected alternative " + x.str; assert false;
         endmatch;
       done
     | x => betas = ([x]) ! betas;

     //| x => println£ "EDLR, unexpected expr " + x.str; assert false;
     endmatch;
     if alphas.len == 0uz do
       outgram = (nt,expr) ! outgram;
     else
       var newntname = fresh();
       var newnt = Recog (Nonterminal[pgram_t] newntname);
       var alts = map (fun (b:list[pgram_t]) => Recog (Seq (b + newnt))) betas;
       outgram =  (nt, Recog (Alt alts)) !  outgram ;
       alts = map (fun (a:list[pgram_t]) => Recog (Seq (a + newnt))) alphas + Recog (Ep
       outgram = (newntname, Recog (Alt alts)) ! outgram;
     done
   done
   return outgram;
  }
```

```
    gen fresh_sym () : string = {
      var n = 1;
    next:>
      yield "_"+n.str;
      ++n;
      goto next;
    }
    // this needs to be global so the algo can be re-applied to the same
    // grammar library
    var fresh = fresh_sym;

    fun direct_left_recursion_elimination (lib:pgramlib_t) =
    {
      return direct_left_recursion_elimination fresh lib;
    }

    fun make_seq (a:pgram_t) (b:list[pgram_t]) =>
      match a with
      | Recog (Seq a) => Recog (Seq (a + b))
      | _ => Recog (Seq (a ! b))
    ;

    // requires one entry per non-terminal, sorted for performance
    // must be in form Alt (Seq (nt, ...)) or Seq (nt, ...) or sym
    // right is the original grammar which i scans thru
    // left is the modified grammar for j = 1 to n -1
    // each recursion advances i one step

    fun left_recursion_elimination_step
      (fresh:1->string)
      (var left:pgramlib_t)
      (var right:pgramlib_t)
    =
    {
       match right with
       | Empty => return left;
       | (rnt,rdfn) ! tail => // A_i
println$ "left_recursion_elimination considering nonterminal A_i=" +rnt;
       var rprods =
         match rdfn with
         | Recog (Alt alts) => alts
         | _ => ([rdfn])
       ;

       var toremove = Empty[int];
       var toadd = Empty[pgram_t];
       match lnt,ldfn in left do // A_j = 1 to i - 1
println$ "  left_recursion_elimination considering nonterminal A_j=" +lnt;
         var lprods =
           match ldfn with
           | Recog (Alt alts) => alts
           | _ => ([ldfn])
         ;
         var counter = -1;
         for rprod in rprods do // A_i = A_j alpha
println$ "    checking if " + rnt + " = " + rprod.str + " has left corner A_j=" + lnt;
```

11.1 Grammar Syntax

```
syntax parser_syn
{
  priority
    palt_pri <
    pseq_pri <
    patom_pri
  ;

  stmt := plibrary =># "_1";

  plibrary := "gramlib" sname "{" plibentry* "}" =>#
    """
    (let*
      (
        (tup `(ast_tuple ,_sr ,_4))
        (v `(ast_apply ,_sr (,(nos "list") ,tup)))
      )
      `(ast_var_decl ,_sr ,_2 ,dfltvs none (some ,v))
    )
    """
  ;

  plibentry := sname "=" pexpr[palt_pri] ";" =>#
  """`(ast_tuple ,_sr (,(strlit _1) ,_3))""";

  sexpr := "parser" "(" pexpr[palt_pri] ")" =># "_3";

  private pexpr[palt_pri] := "|"? pexpr[>palt_pri] ("|" pexpr[>palt_pri])+ =>#
    """`(ast_apply ,_sr (
      ,(qnoi 'Parser_synlib 'ALT)
      (ast_apply ,_sr (,(noi 'list) ,(cons _2 (map second _3))))))""""
  ;

  private pexpr[pseq_pri] := pexpr[>pseq_pri] (pexpr[>pseq_pri])+ =>#
    """`(ast_apply ,_sr (
      ,(qnoi 'Parser_synlib 'SEQ)
      (ast_apply ,_sr (,(noi 'list) ,(cons _1 _2)))))""""
  ;

  private pexpr[patom_pri] := "(" pexpr[palt_pri] ")" =># "_2";

  private pexpr[patom_pri] := String =>#
    """`(ast_apply ,_sr ( ,(qnoi 'Parser_synlib 'STR) ,_1)) """
  ;

  private pexpr[patom_pri] := "#EPS" =>#
    """`(ast_apply ,_sr ( ,(qnoi 'Parser_synlib 'EPS) ())) """
  ;

  private pexpr[patom_pri] := sname=>#
    """`(ast_apply ,_sr ( ,(qnoi 'Parser_synlib 'NT) ,(strlit _1))) """
  ;

  private pexpr[patom_pri] := "{" pexpr "}" =># "_2";
```

11.2 Simplified Constructors

```
class Parser_synlib
{
  open Parsers;
  open Grammars;
  fun NT (s:string) => Recog (Nonterminal [pgram_t] s);
  fun TERM (s:string, r:Recognisers::recog_t) => Recog (Terminal [pgram_t] (s,r));
  fun STR (s:string) => Recog (Terminal [pgram_t] (s, (Recognisers::match_string s)));
  fun REDUCE (s:string, n:int) => Action[pgram_t] (Reduce (s,n));
  fun BINOP(s:string) => Action[pgram_t] (Sequence ([Swap, Drop 1, (Reduce (s,2))]));
  fun SWAP () => Action[pgram_t] (Swap);
  fun DROP (n:int) => Action[pgram_t] (Drop n);
  fun ALT (ls: list[pgram_t]) => Recog (Alt[pgram_t] ls);
  fun SEQ (ls: list[pgram_t]) => Recog (Seq[pgram_t] ls);
  fun EPS () => Recog (Epsilon[pgram_t]);
}
```

Part I

Appendices

Chapter 12

Coroutine Semantics

12.1 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.

12.1.1 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.

12.1.2 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.

12.1.3 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.

12.2 Abstract State

12.2.1 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

$$H = \{(f, c) \mid f \in H_c\} \quad \text{Hunger} \quad (12.1)$$

$$B = \{(f, c) \mid f \in B_c\} \quad \text{Blockage} \quad (12.2)$$

$$\mathcal{F}_\mathcal{H} = \{f \mid \exists c.(f, c) \in \mathcal{H}\} \quad \text{Hungry Fibres} \quad (12.3)$$

$$\mathcal{F}_\mathcal{B} = \{f \mid \exists c.(f, c) \in \mathcal{B}\} \quad \text{Blocked Fibres} \quad (12.4)$$

$$\mathcal{C}_\mathcal{H} = \{c \mid \exists f.(f, c) \in \mathcal{H}\} \quad \text{Hungry Channels} \quad (12.5)$$

$$\mathcal{C}_\mathcal{B} = \{c \mid \exists f.(f, c) \in \mathcal{B}\} \quad \text{Blocked Channels} \quad (12.6)$$

$$\mathcal{E} = \{c \mid |\mathcal{H}_c = \mathcal{B}_c = \emptyset\} \quad \text{Empty Channels} \quad (12.7)$$

12.2.2 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
```

12.3 Operations

12.3.1 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.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{12.8}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{f\} \\ f, \mathcal{A}_k \cup \{R_s\} \end{cases} \tag{12.9}$$

where the choice between the two cases is indeterminate.

12.3.2 Run

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

$$k \leftarrow k + 1 \tag{12.10}$$

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 $\emptyset$.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{12.11}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow f, \emptyset \tag{12.12}$$

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.

$$k \leftarrow k - 1 \tag{12.13}$$

12.3.3 Create channel

A function which creates a channel.

$$\mathcal{C} \leftarrow \mathcal{C} \cup \{c\} \tag{12.14}$$
$$\mathcal{E} \leftarrow \mathcal{E} \cup \{c\} \tag{12.15}$$

where c is a fresh channel.

12.3.4 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.

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{12.16}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{12.17}$$

   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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{12.18}$$

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

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{12.19}$$
$$\tag{12.20}$$

   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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if} A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A_k \end{cases} \tag{12.21}$$

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

$$\mathcal{B} \leftarrow \mathcal{B} \setminus (w, c) \tag{12.22}$$

   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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{w\} \\ w, \mathcal{A}_k \cup \{R\} \end{cases} \tag{12.23}$$

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

12.3.5 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.

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{12.24}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{12.25}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{12.26}$$

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

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{12.27}$$

(12.28)

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{12.29}$$

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

$$\mathcal{H} \leftarrow \mathcal{H} \setminus (r, c) \tag{12.30}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{r\} \\ r, \mathcal{A}_k \cup \{R\} \end{cases} \tag{12.31}$$

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

12.3.6 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.

12.3.7 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.

12.4 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.

12.5 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.

12.6 Continuation Structure

12.6.1 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

12.6.2 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.

12.7 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.

12.8 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).

12.9 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.

12.9.1 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.

12.9.2 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.

12.9.3 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.

12.9.4 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.

12.10 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.

12.10.1 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.

12.11 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.

Chapter 13

Installing Felix

13.1 Prerequisites

Ocaml optimising native code compiler 4.03 or higher. Python 3.4 or higher. Either g++ or clang++ with C++11.

13.1.1 Optional

SDL2, GMP.

13.1.2 Unix

For general Unix, Linux, Windows Subsystem for Linux, Cygwin: Binutils, GNUmake. If both clang++ and g++ are installed, the bootstrap will use g++.

```
. buildscript/linuxsetup.sh
make
sudo make install #optional
```

13.1.3 OSX

Binutils, GNUmake. If both clang++ and g++ are installed the bootstrap will use clang++.

```
. buildscript/osxsetup.sh
make
sudo make install #optional
```

13.1.4 Windows

Binutils, nmake. You also need Ocaml and MSVC++. Please examine the file `appveyor.yml` which does a complete build on the Appveyor continuous integration server. If you have an uptodate Windows 10 with Bash for Windows (WSL), that may be easier to use.

List of Listings

Code Index

General Index