Coroutines

John Skaller

March 21, 2020

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.

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.

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.

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.

2 Abstract State

2.1 State Data by Sets

A fibration system consists of

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

and the following relations:

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

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

We define the relation

$$H = \{(f, c) \mid f \in H_c\} \quad \text{Hunger} \tag{1}$$
$$B = \{(f, c) \mid f \in B_c\} \quad \text{Blockage} \tag{2}$$
$$\mathcal{F}_\mathcal{H} = \{f \mid \exists c.(f, c) \in \mathcal{H}\} \quad \text{Hungry Fibres} \tag{3}$$
$$\mathcal{F}_\mathcal{B} = \{f \mid \exists c.(f, c) \in \mathcal{B}\} \quad \text{Blocked Fibres} \tag{4}$$
$$\mathcal{C}_\mathcal{H} = \{c \mid \exists f.(f, c) \in \mathcal{H}\} \quad \text{Hungry Channels} \tag{5}$$
$$\mathcal{C}_\mathcal{B} = \{c \mid \exists f.(f, c) \in \mathcal{B}\} \quad \text{Blocked Channels} \tag{6}$$
$$\mathcal{E} = \{c \mid |\mathcal{H}_c = \mathcal{B}_c = \emptyset\} \quad \text{Empty Channels} \tag{7}$$

2.2 State Data by ML

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

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

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

fibre = (current: continuation)

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

3 Operations

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.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{8}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{f\} \\ f, \mathcal{A}_k \cup \{R_s\} \end{cases} \tag{9}$$

where the choice between the two cases is indeterminate.

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.

$$k \leftarrow k + 1 \tag{10}$$

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 $\emptyset$.

$$\mathcal{F} \leftarrow \mathcal{F} \cup \{f\} \tag{11}$$

where f is a fresh fibre and

$$R_k, \mathcal{A}_k \leftarrow f, \emptyset \tag{12}$$

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.

$$k \leftarrow k - 1 \tag{13}$$

3.3 Create channel

A function which creates a channel.

$$\mathcal{C} \leftarrow \mathcal{C} \cup \{c\} \tag{14}$$
$$\mathcal{E} \leftarrow \mathcal{E} \cup \{c\} \tag{15}$$

where c is a fresh channel.

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.

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{16}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{17}$$

   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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{18}$$

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

$$\mathcal{H} \leftarrow \mathcal{H} \cup \{(r, c)\} \tag{19}$$
$$\tag{20}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if} A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A_k \end{cases} \tag{21}$$

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

$$\mathcal{B} \leftarrow \mathcal{B} \setminus (w, c) \tag{22}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{w\} \\ w, \mathcal{A}_k \cup \{R\} \end{cases} \tag{23}$$

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

3.5 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.

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{24}$$
$$\mathcal{E} \leftarrow \mathcal{E} \setminus \{c\} \tag{25}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{26}$$

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

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{(w, c)\} \tag{27}$$
$$\tag{28}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} \epsilon, \mathcal{A}_k & \text{if } A_k = \emptyset \\ a, \mathcal{A}_k \setminus \{a\} & \text{some } a \in A \end{cases} \tag{29}$$

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

$$\mathcal{H} \leftarrow \mathcal{H} \setminus (r, c) \tag{30}$$

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.

$$R_k, \mathcal{A}_k \leftarrow \begin{cases} R_k, \mathcal{A}_k \cup \{r\} \\ r, \mathcal{A}_k \cup \{R\} \end{cases} \tag{31}$$

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

3.6 Reachability

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

If a channel is known to reachable fibre, it is also reachable. A channel may be known because its address is stored in the local data of a continuation of a fibre, or, it is reachable via some object which can be reached from local data. The exact rules are programming language dependent.

Each fibre associated with a reachable channel is reachable.

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

Unreachable fibres and channels are automtically garbage collected.

3.7 Elimination

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

A fibre may become unreachable in three ways.

3.7.1 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.

3.7.2 Starvation

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

3.7.3 Blockage

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

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.

5 Fibre Structure

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

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

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

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

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

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

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

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

6 Continuation Structure

6.1 Continuation Data

A continuation has associated with it the following data:

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

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

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

6.2 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.

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.

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).

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.

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.

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.

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.

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.

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.

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.

11 Felix Implementation

The following functions and procedures are provided in Felix:

```
    spawn_fthread: (1 -> 0) -> 0;
    run: (1 -> 0) -> 0;
```

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

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

The following shorthand types are available:

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

More advanced typing exploiting channel capabilities are discussed later.

Simple example program:

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

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

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

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

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

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

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

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

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

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

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

This usually occurs because some other fibre is statically capable of resolving the lockup, but does not do so dynamically. To prevent livelocks, variables holding channel values to which no I/O will occur dynamically should also go out of scope.