Asynchronous Effect Handling 2

Leo Poulson

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2020

Abstract

Features for asynchronous programming are commonplace in the programming languages of today, allowing programmers to issue tasks to run on other threads and wait for the results to come back later. This is particularly useful for programs like web programs, etc...

In this thesis we show how asynchronous programming can be very easily accomodated in a language with existing support for effect handlers. We show how, with a small change to the language implementation, truly asynchronous programming with pre-emptive concurrency is achieved.

Acknowledgements

thanks!

Table of Contents

Chapter 1

Introduction

Effects, such as state and nondeterminism, are pervasive when programming; for a program to do anything beyond compute a pure mathematical function, it must interact with the outside world, be this to read from a file, make some random choice, or run concurrently with another program. Algebraic effects and their handlers (Plotkin and Pretnar [2013]) are a novel way to encapsulate, reason about and specify computational effects in programming languages. For instance, a program that reads from and writes to some local state can utilise the State effect, which supports two *operations*; get and put. A handler for the State effect gives a meaning to these abstract operations.

Programming with effects is increasingly popular, references, ...

Traditional effect handling is *synchronous*; when an operation is invoked, the rest of the computation pauses whilst the effect handler performs the requisite computation and then resumes the original caller. For many effects, this blocking behaviour is not a problem; the handler usually returns quickly, and the user does not notice anything. However, not every possible computational effect behaves like this. Consider an effect involving a query to a remote database. We might not want to block the rest of the computation whilst we perform this, as the query might take a long time; this case is even stronger if we do not immediately want the data. To support this kind of behaviour, we need to be able to invoke and handle effects in an asynchronous, non-blocking manner.

In this project we investigate the implementation and applications of asynchronous effects. Our lens for this is the language Frank (Convent et al. [2020]), a functional programming designed with effect handlers at its core. We follow the design of Æff (Ahman and Pretnar [2020]), a small programming language designed around asynchronous effects but supporting little else. We show how, with a small change to the semantics of Frank, we can recreate the asynchronous effect handling behaviour of Æff whilst enjoying the benefits of traditional effect handlers.

Frank is a well-suited language for an asynchronous effects library, especially because of the fine-grained control over suspended computations, making it very easy to treat code as data. Despite this, our approach does not use any specific Frank features; furthermore, the changes made to the semantics of Frank are easily recreateable. It is our hope that these methods could be recreated in another language equipped with first-class effect handlers. Effect handlers have already proven to make complicated control flow easy to implement (**refs**), and our work further cements this viewpoint.

1.1 Related Work

Asynchronous programming with effect handlers is a fairly nascent field. Koka (Leijen [2014]) is a programming language with built-in effect handlers and a Javascript backend. Leijen later shows us how Koka can naturally support asynchronous programming (Leijen [2017]). The asynchronous behaviour relies on offloading asynchronous tasks with a setTimeout function supplied by the NodeJS backend.

Multicore OCaml (**ref**) also supports asynchronous programming through effect handling. They handle effects and signals, which can be received asynchronously, and show how to efficiently and safely write concurrenct systems programs. However, in a similar way to Koka, the asynchrony relies on the operating system supplying operations, such as setSignal and timer signals.

A problem shared by both Koka and Multicore OCaml is they have no support for *user-defined* asychronous effects; the asynchronous signals that can be received are predefined. This problem is solved by Æff (Ahman and Pretnar [2020]), a small language built around asynchronous effect handling. Ahman and Pretnar approach the problem of asynchrony from a different perspective, by decoupling the invocation of an effect from its resumption with the handled value. When an effect is invoked the rest of the computation is not blocked whilst the handler is performed. Programs may then install interrupt handlers that dictate how to act on receipt of a particular interrupt. To recover synchronous behaviour, these interrupt handlers can be awaited; this will block the rest of the code until the interrupt is received.

TODO: Section on the expressivity?

We design our system around Æff, embedding its behaviour into Frank. We show that we can easily recover the behaviour of Æff when equipped with effect handlers, and show that asynchronous effects can still be used in conjunction with traditional, synchronous effects.

1.2 Contributions

Asynchronous Effects Library We present a library for programming with asynchronous effects in the style of Æff, built in Frank. We show how a complex system can be expressed concisely and elegantly when programming in a language with effect handlers, further cementing the case for effects as a foundation for concurrent programming.

TODO: Rewrite the end of this; slightly messy

.

Pre-emptive Concurrency We show how, by making a small change to the operational semantics of Frank, we achieve pre-emptive concurrency; that is, the suspension of running threads *without* co-operation. It is our hope that this change is simple enough to be transferrable to other languages.

Examples We also deliver a set of examples of the uses of asynchronous effects, and show how they have benefits to other models.

1.3 Structure

In Chapter 2 we give an introduction to programming with effects in Frank. We skip over some unneeded (and previously well-covered) parts of the language, such as adaptors, in the interests of time.

In Chapter 3 we give the formalisation of Frank. Again, we skip over extraneous details which can be seen in past work (Convent et al. [2020]), opting to only describe the parts needed to understand the changes to the semantics for the following chapter.

In Chapter 4 we show how by making a small change to the semantics of Frank we yield pre-emptible threads; that is, we can interrupt a function in the same 'co-operative' style but without co-operation

In Chapter 5 we describe the implementation of our asynchronous effect handling library in Frank. In Chapter 6 we give examples of the new programs that become easily expressible when combined with the changes made in Chapter 4.

In Chapter 7 we conclude.

Chapter 2

Programming in Frank

Frank is a functional programming language, designed with the use of algebraic effects at its heart. As such, Frank has an effect type system used to track which effects a computation may use.

Frank offers very fine-grained control over computations. It clearly distinguishes between computation and value, and offers *multihandlers* to carefully control when computations are evaluated. This combined with effect handling provides a very rich foundation for expressing complex control structures.

In this chapter we introduce Frank, and show why it is so well-suited to our task. We assume some familiarity with typed functional programming, and skip over some common features of Frank — algebraic data types, pattern matching, etc. — so we can spend more time with the novel, interesting parts.

2.1 Types, Values and Operators

Frank types are distinguished between *effect types* and *value types*. Value types are the standard notion of type; effect types are used to describe where certain effects can be performed and handled. Value types are further divided into traditional data types, such a `Bool`, `List X`, and *computation types*.

A computation type $\mathtt{X}_1$ `-> ... ->` $\mathtt{X}_m$ `-> [`$\mathtt{I}_1$`, ...,` $\mathtt{I}_n$`]Y` expresses that we take m arguments and return a value of type `Y`. The return type expresses the *ability* the computation needs access to, being a list of n *interface* instances. An interface is a collection of *commands* which are offered to the computation.

Thunks are then the special case of an n-ary function that takes 0 arguments. We can evaluate them — performing the suspended computation — with the 0-ary sequence of arguments, denoted `!`. Computations are suspended by wrapping them in braces.

This gives us fine-grained control over when we want to evaluate computations. For instance, we might want the sequential composition operator `snd` (also commonly known as the semicolon operator);

```
snd : {X -> Y -> Y}
snd x y = y
```

Frank is a left-to-right call-by-value language; arguments to functions are evaluated from left-to-right, until they become a value. In the case of `snd`, `x` is first evaluated, then `y`, which is finally returned. Compare this to `if`;

```
if : {Bool -> {X} -> {X} -> X}
if true  yes no = yes!
if false yes no = no!
```

The branches are given as thunks, where a single thunk is evaluated depending on the condition. If we did not take this approach both cases would be evaluated, which is clearly not the intended semantics of `if`. Frank's distinction between computation and value make controlling evaluation simple and pleasing.

2.2 Effects and Effect Handling

Interfaces and Operations Frank encapsulates effects through *interfaces*, which offer *commands*. For instance, the `State` effect (interface) offers two operations (commands), `get` and `put`. In Frank, this translates to

```
interface State X = get : X
                  | put : X -> Unit

interface RandInt = random : Int
```

The type signatures of the operations mean that `get` is a 0-ary operation which is *resumed* with a value of type `X`, and `put` takes a value of type `X` and is resumed with `unit`. Computations get access to an interface's commands by including them in the *ability* of the program. Commands are invoked just as normal functions;

```
xplusplus : {[State Int] Unit}
xplusplus! = put (get! + 1)
```

This familiar program increments the integer in the state by 1.

Handling Operations A handler for a specific interface can also pattern match on the `operations` that are performed, and not just the values that can be returned. As an example, consider the canonical handler for the `State S` interface.

```
runState : {<State S> X -> S -> X}
runState <get -> k>    s = runState (k s) s
runState <put s -> k> _ = runState (k unit) s
runState x             _ = x
```

Observe that the type of `runState` contains `<State S>`, called an *adjustment*. This expresses that the first argument can perform commands in the `State S` interface, and that `runState` must handle these commands if they occur.

Computation Patterns The second and third lines specify how we handle `get` and `put` commands. Observe that we use a new type of pattern, called a *computation pattern*; these are made up of a command and some arguments (which are also values), plus the continuation of the calling code. The types of arguments and the continuation are determined by the interface declaration and the type of the handler; for instance, in `<get -> k>` the type of `k` is `{S -> [State S] X}`. The continuation can then perform more `State S` effects. This differs to some other implementations of effect handling languages

TODO: Add references

where the handlers are *deep*, meaning the continuation has been re-handled by the same handler automatically.

Effect Forwarding Effects that are not handled by a particular handler are left to be forwarded up to the next highest one. For instance, we might want to write a random number to the state;

```
xplusrand : {[State Int, RandomInt] Unit}
xplusrand! = put (random!)
```

We then have to handle both the `State Int` and `Random` effect in this computation. Of course, we could just define one handler for both effects; however in the interests of *modularity* we want to define two different handlers for each effect and *compose* them. We can reuse the same `runState` handler from before, and define a new handler for `RandomInt`;

```
runRand : {Int -> <RandomInt> X -> X}
runRand seed <random -> k> = runRand (mod (seed + 7) 10) (k seed)
runRand _ x = x
```

And compose them in the comfortable manner, by writing `runRand (runState xplusrand !)`.

TODO: Maybe show example of how the order of composition can change the ending semantics — a la state + aborting

Top-Level Effects Some effects need to be handled outside of pure Frank, as Frank is not expressive or capable enough on its own. Examples are console I/O, web requests, and ML-style state cells. These effects will pass through the whole stack of handlers up to the top-level, at which point they are handled by the interpreter.

Implicit Effect Polymorphism Consider the type of the well-known function `map` in Frank;

```
map : {{X -> Y} -> List X -> List Y}
map f [] = []
map f (x :: xs) = (f x) :: (map f xs)
```

One might expect that the program `map {_ -> random!} [1, 2, 3]` would give a type error; we are mapping a function of type `{Int -> [RandomInt] Int}`, which does not match the argument type `{X -> Y}`. However, Frank uses a shorthand for *implicit effect variables*. The desugared type of `map` is actually

```
map : {{X -> [ε|] Y} -> List X -> [ε|] List Y}
```

This type expresses that whatever the ability is of `map f xs` will be offered to the element-wise operator `f`. As such, the following typechecks;

```
writeRand : {List Int -> [RandomInt] List Int}
writeRand xs = map {_ -> random!} xs
```

TODO: Talk about deliberately stopping this

TODO: Talk about what the bar means.

A similar thing happens in interface declarations. We might define the `Choose` effect, which non-deterministically asks for one of two computations to be picked for it to continue with;

```
interface Choose X =
    choose : {[Choose X] X} -> {[Choose X] X} -> X
```

This definition desugars to

```
interface Choose X [ε] =
    choose : {[ε| Choose X] X} -> {[ε| Choose X] X} -> X
```

Once again, an implicit effect variable is inserted in every ability available.

Synchronicity and Conversations Observe how the interaction between the effect invoking function and the handler of this effect becomes like a conversation; the caller asks the handler for a response to an operation, and the caller will then wait, blocking, for a response. We characterise this as *synchronous* effect handling; the operation caller and the handler synchronise to communicate.

But what if we want to make a request for information, then do something else, then pick up the result later when we need it? This is the canonical example of asynchronous programming. It is not as simple as just invoking our e.g. `getRequest` effect; computation would block once this is invoked, meaning we are stuck waiting for the request to return. This sort of behaviour is exactly what we want to explore in this project.

Multihandlers Recall that in Frank pure functions are just the special case of handlers that handle no effects. Naturally, this notion extends to the *n*-ary case; we can handle multiple effects from different sources are once. Handlers which handle multiple effects simultaneously are unsurprisingly called *multihandlers*. This lets us write functions such as `pipe` (example due to Convent et al. [2020]);

```
1 interface Send X = send : X -> Unit
2 interface Receive X = receive : X
3
4 pipe : {<Send X>Unit -> <Receive X>Y -> [Abort]Y}
5 pipe <send x -> s> <receive -> r> = pipe (s unit) (r x)
6 pipe <_> y = y
7 pipe unit <_> = abort!
```

Line 5 states that `pipe` will handle all instances of the `Send` effect in the first argument, all instances of the `Receive` effect in the second, and might perform `Abort` commands along the way. The matching clauses are also new to the reader; line 6 implements the communication between the two functions. We reinvoke `pipe`, passing the payload `x` of `send` to the continuation of `r`. Lines 7 and 8 make use of the *catch-all* pattern, <m>. This will match the invocation of any effect that is handled by that argument, or a value, binding this to `m`. In line 7, the catchall pattern matches either a `send` command or a value; in this case, the receiver has produced a value, so we can return that. In line 8 <_> matches either a value or a `receive`; but it must be a `receive` command, as the value case would have been caught above. The `abort` command is then invoked, as this is erroneous. A recovery strategy can be implemented by a handler for `Abort`.

Polymorphic Commands As well as having polymorphic interfaces, such as `State X`, parametrised by e—.g. the data stored in the state, Frank supports polymorphic *commands*. These are commands which can be instantiated for any type. An example is ML-style references, realised through the `RefState` interface;

```
interface RefState = new X   : X -> Ref X
                   | read X  : Ref X -> X
                   | write X : Ref X -> X -> Unit
```

For instance, `new X` can be instantiated by supplying a value as an argument. A `Ref X` cell is then returned as answer.

2.3 Case Study: Cooperative Concurrency

Effect handlers have proved to be useful abstractions for concurrent programming (Dolan et al. [2015, 2017], Hillerström [2016]). This is partly because the invocation of an operation not only offers up the operation's payload, but also the *continuation* of the calling computation. In Frank, these continuations are first-class. The handler for this operation is then free to do what it pleases with the continuation. For many effects, such as `getState`, nothing interesting happens to the continuation and it is just resumed immediately. But these continuations are first-class; they can resumed, but also stored elsewhere or even thrown away.

We illustrate this with some examples of concurrency in this section.

2.3.1 Simple Scheduling

We introduce some simple programs and some scheduling multihandlers, to demonstrate how subtly different handlers generate different scheduling strategies.

```
interface Yield = yield : Unit

words : {[Console, Yield] Unit}
words! = print "one "; yield!; print "two "; yield!; print "three ";
    yield!

numbers : {[Console, Yield] Unit}
numbers! = print "1 "; yield!; print "2 "; yield!; print "3 "; yield
   !
```

First note the simplicity of the `Yield` interface; we have one operation supported, which looks very boring; the operation `yield!` will just return unit. It is the way we *handle* yield that is more interesting. These two programs will print some information out and yield inbetween each print operation.

```
1 schedule : {<Yield> Unit -> <Yield> Unit -> Unit}
2 schedule <yield -> m> <yield -> n> = schedule (m unit) (n unit)
3 schedule <yield -> m> <n> = schedule (m unit) n!
4 schedule <m> <yield -> n> = schedule m! (n unit)
5 schedule _ _ = unit
```

When we run `schedule words! numbers!` we read `one 1 two 2 three 3 unit` from the console. What happened? First `words` is evaluated until it results in a `yield` command. Recall that Frank is a left-to-right call-by-value language; at this point, we start evaluating the second argument, `numbers`. This again runs until a `yield` is performed, where we give control again to the scheduler. Now that both arguments are commands or values we can proceed with pattern matching; the first case matches and we resume both threads, handling again. This process repeats until one thread evaluates to `unit`. In this way, we can imagine multihandler arguments as running in parallel and then *synchronising* when pattern matching is performed.

If we omit the first line of pattern matching (line 2) we get quite a different result; the console output would be `one 1 two three 2 3`. This is because both threads are evaluated until they are `yield` invocations, printing out `one 1`; but then only the first thread is resumed. We then evaluate `words` until it becomes a `unit` value. At this point the patterns on line 4 match, as the catchall pattern `<m>` will match commands or values. We then evaluate `numbers` until this is also `unit`.

2.3.2 Forking New Processes

We can make use of Frank's higher-order effects to dynamically create new threads at runtime. We strengthen the `Yield` interface by adding a new operation `fork`;

```
interface Co = fork : {[Co] Unit} -> Unit
             | yield : Unit
```

The type of `fork` expresses that `fork` takes a suspended computation that can perform further `Co` effects, and returns unit when handled. We can now run programs that allocate new threads at runtime, such as the below

```
forker : {[Console, Co [Console]] Unit}
forker! = print "Starting! ";
```

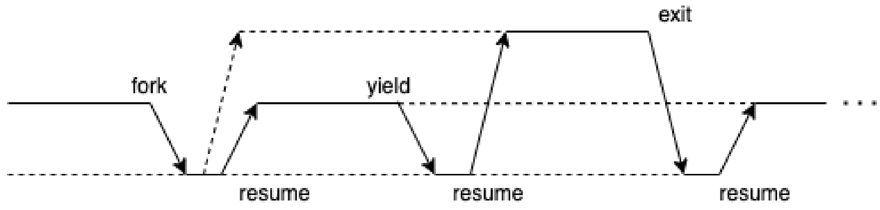

Figure 2.1: Breadth-First scheduling

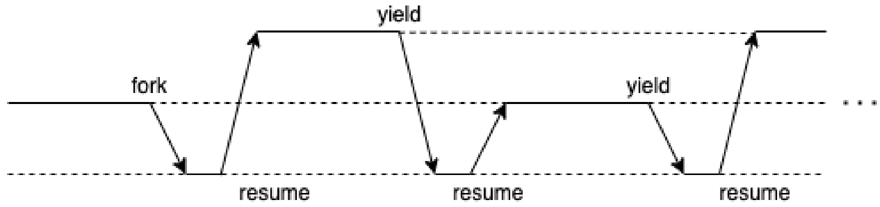

Figure 2.2: Depth-First scheduling

```
fork {print "one "; yield!; print "two "};
fork {print "1 "; yield!; print "2 "};
exit!
```

We can now choose a strategy for handling **fork** operations; we can either lazily run them, by continuing our current thread and then running them, or eagerly run them, suspending the currently executing thread and running the forked process straight away. The handler for the former, breadth-first style of scheduling, is;

```
scheduleBF : {<Co> Unit -> [Queue Proc] Unit}
scheduleBF <fork p -> k> = enqueue {scheduleBF (<Queue> p!)};
                           scheduleBF (k unit)
scheduleBF <yield -> k>  = enqueue {scheduleBF (k unit)};
                           runNext!
scheduleBF unit          = runNext!
```

where the operations **enqueue** and **runNext** are offered by the **Queue** effect. We have to handle the computation **scheduleBF forker!** with a handler for **Queue** effects afterwards. We can abstract over different queue handlers for even more possible program combinations. Moreover, notice how concisely we can express the scheduler; this is due to the handler having access to te continuation of the caller, and treating it as a first-class object that can be stored elsewhere. We can see a diagram of how **scheduleBF** treats continuations in Figure 2.1, and a similar diagram of how the depth-first handling differs in Figure 2.2.

Chapter 3

Formalisation of Frank

The formalisation of the Frank language has been discussed at length in previous work (Convent et al. [2020]). However, in order to illustrate changes made to the language in this work, we explain some of the relevant parts of the language.

(data types)	D
(value type variables)	X
(effect type variables)	E
(value types)	$A, B ::= D\,\overline{R}$
	$\mid\ \{C\} \mid X$
(computation types)	$C ::= \overline{T \to} G$
(argument types)	$T ::= \langle\Delta\rangle A$
(return types)	$G ::= [\Sigma]A$
(type binders)	$Z ::= X \mid [E]$
(type arguments)	$R ::= A \mid [\Sigma]$
(polytypes)	$P ::= \forall\overline{Z}.A$
(interfaces)	I
(term variables)	x, y, z, f
(instance variables)	s, a, b, c
(seeds)	$\sigma ::= \emptyset \mid E$
(abilities)	$\Sigma ::= \sigma \mid \Xi$
(extensions)	$\Xi ::= \iota \mid \Xi, I\,\overline{R}$
(adaptors)	$\Theta ::= \iota \mid \Theta, I(S \to S')$
(adjustments)	$\Delta ::= \Theta \mid \Xi$
(instance patterns)	$S ::= s \mid S\ a$
(kind environments)	$\Phi, \Psi ::= \cdot \mid \Phi, Z$
(type environments)	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, f : P$
(instance environments)	$\Omega ::= s : \Sigma \mid \Omega, a : I\,\overline{R}$

Figure 3.1: Types

Types Value types are either datatypes instantiated with type arguments $D\,\overline{R}$, thunked computations $\{C\}$, or value type variables X. Computation types are of the form

$$C = \langle\Theta_1 \mid \Xi_1\rangle A_1 \to \cdots \to [\Sigma]B$$

where a computation of type C handles effects in Ξ_i or pattern matches in A_i on the i-th argument and returns a value of type B. C may perform effects in ability Σ along

$$
\begin{array}{ll}
\text{(constructors)} & k \\
\text{(commands)} & c \\
\text{(uses)} & m ::= x \mid f\ \overline{R} \mid m\ \overline{n} \mid \uparrow(n : A) \\
\text{(constructions)} & n ::= \downarrow m \mid k\ \overline{n} \mid c\ \overline{R}\ \overline{n} \mid \{e\} \\
& \quad \mid\ \mathbf{let}\ f : P = n\ \mathbf{in}\ n' \mid \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n \\
& \quad \mid\ \langle\Theta\rangle\ n \\
\text{(computations)} & e ::= \overline{\overline{r} \mapsto n} \\
\text{(computation patterns)} & r ::= p \mid \langle c\ \overline{p} \to z\rangle \mid \langle x\rangle \\
\text{(value patterns)} & p ::= k\ \overline{p} \mid x
\end{array}
$$

Figure 3.2: Terms

the way. The i-th argument to C can perform effects in Σ adapted by adaptor Θ_i and augmented by extension Ξ_i.

An ability Σ is an extension Ξ plus a seed, which can be closed ($\emptyset$) or open E. This lets us explicitly choose whether a function can be effect polymorphic, as discussed earlier. An extension Ξ is a finite list of interfaces.

We omit details on adaptors as they are present in previous work. The same goes for the typing rules, which we do not change.

Terms Frank uses bidirectional typing (Pierce and Turner [2000]); as such, terms are split into *uses* whose types are inferred, and *constructions*, which are checked against a type. Uses are monomorphic variables x, polymorphic variable instantiations $f\ \overline{R}$, applications $m\ \overline{n}$ and type ascriptions $\uparrow(n : A)$. Constructions are made up of uses $\downarrow m$, data constructor instances $k\ \overline{n}$, suspended computations $\{e\}$, let bindings $\mathbf{let}\ f : P = n\ \mathbf{in}\ n'$, recursive let $\mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n$ and adaptors $\langle\Theta\rangle\ n$. We can inject a use into a construction and vice versa ($\downarrow$, $\uparrow$); in real Frank code these are not present.

Computations are produced by a sequence of pattern matching clauses. Each pattern matching clause takes a sequence $\overline{r}$ of computation patterns. These can either be a request pattern $\langle c\ \overline{p} \to z\rangle$, a catch-all pattern $\langle x\rangle$, or a standard value pattern p. Value patterns are made up of data constructor patterns $k\ \overline{p}$ or variable patterns x.

Runtime Syntax The operational semantics uses the runtime syntax of Figure 3.3. Uses and constructions are further divided into those which are values. Values are either variable or datatype instantiations, or suspended computations. We also declare a new class of *normal forms*, to be used in pattern binding. These are either construction values or *frozen commands*, $\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil$. Frozen commands are used to capture a

$$
\begin{array}{lrl}
\text{(uses)} & m ::= & \cdots \mid \lceil \mathcal{E}[c\,\overline{R}\,\overline{w}] \rceil \\
\text{(constructions)} & n ::= & \cdots \mid \lceil \mathcal{E}[c\,\overline{R}\,\overline{w}] \rceil \\
\text{(use values)} & u ::= & x \mid f\,\overline{R} \mid \uparrow(v : A) \\
\text{(non-use values)} & v ::= & k\,\overline{w} \mid \{e\} \\
\text{(construction values)} & w ::= & \downarrow u \mid v \\
\text{(normal forms)} & t ::= & w \mid \lceil \mathcal{E}[c\,\overline{R}\,\overline{w}] \rceil \\
\text{(evaluation frames)} & \mathcal{F} ::= & [\,]\;\overline{n} \mid u\;(\overline{t},[\,],\overline{n}) \mid \uparrow([\,] : A) \\
 & \mid & \downarrow[\,] \mid k\;(\overline{w},[\,],\overline{n}) \mid c\,\overline{R}\;(\overline{w},[\,],\overline{n}) \\
 & \mid & \mathbf{let}\; f : P = [\,]\; \mathbf{in}\; n \mid \langle \Theta \rangle\; [\,] \\
\text{(evaluation contexts)} & \mathcal{E} ::= & [\,] \mid \mathcal{F}[\mathcal{E}]
\end{array}
$$

Figure 3.3: Runtime Syntax

continuation's *delimited continuation*. As soon as a command is invoked it becomes frozen. The entire rest of the computation around the frozen command then also freezes (in the same way that water behaves around ice), until we reach a handler for the frozen command.

Finally we have evaluation contexts, which are sequences of evaluation frames. The interesting case is $u\ (\overline{t}, [\ ], \overline{n})$; it is this that gives us left-to-right call-by-value evaluation of multihandler arguments.

Operational Semantics Finally, the operational semantics are given in Figure 3.4.

The essential rule here is R-HANDLE. This relies on a new relations regarding *pattern binding* (Figure 3.5). $r : T \leftarrow t \dashv\!\!_{[\Sigma]}\ \theta$ states that the computation pattern r of type T at ability Σ matches the normal form t yielding substitution θ. The index k is then the index of the earliest line of pattern matches that all match. The conclusion of the rule states that we then perform the substitutions $\overline{\theta}$ that we get on the return value n_k to get our result. This is given type B.

R-ASCRIBE-USE and R-ASCRIBE-CONS remove unneeded conversions from use to construction. R-LET and R-LETREC are standard. R-ADAPT shows that an adaptor applied to a value is the identity.

We have several rules regarding the freezing of commands. When handling a command, we need to capture its delimited continuation; that is, the largest enclosing evaluation context that does *not* handle it. R-FREEZE-COMM expresses that invoked commands instantly become frozen; R-FREEZE-FRAME-USE and R-FREEZE-FRAME-CONS show how the rest of the context becomes frozen. These two rules rely on the predicate $\mathcal{E}$ handles c. This is true if the context does indeed handle the com-

$\boxed{m \rightsquigarrow_{\mathrm{u}} m'}$ $\boxed{n \rightsquigarrow_{\mathrm{c}} n'}$ $\boxed{m \longrightarrow_{\mathrm{u}} m'}$ $\boxed{n \longrightarrow_{\mathrm{c}} n'}$

R-HANDLE

$$\frac{k = \min_i \{i \mid \exists \overline{\theta}.(r_{i,j} : \langle \Delta_j \rangle A_j \leftarrow t_j \dashv [\Sigma]\ \theta_j)_j\} \qquad (r_{k,j} : \langle \Delta_j \rangle A_j \leftarrow t_j \dashv [\Sigma]\ \theta_j)_j}{\uparrow(\{((r_{i,j})_j \to n_i)_i\} : \{\overline{\langle \Delta \rangle A \to}\ [\Sigma]B\})\ \overline{t} \rightsquigarrow_{\mathrm{u}} \uparrow((\overline{\theta}(n_k) : B)}$$

R-ASCRIBE-USE

$$\frac{}{\uparrow(\downarrow u : A) \rightsquigarrow_{\mathrm{u}} u}$$

R-ASCRIBE-CONS

$$\frac{}{\downarrow\uparrow(w : A) \rightsquigarrow_{\mathrm{c}} w}$$

R-LET

$$\frac{}{\mathbf{let}\ f : P = w\ \mathbf{in}\ n \rightsquigarrow_{\mathrm{c}} n[\uparrow(w : P)/f]}$$

R-LETREC

$$\frac{\overline{\overline{e = \overline{r} \to n}}}{\mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n' \rightsquigarrow_{\mathrm{c}} n'[\overline{\uparrow(\{\overline{r} \to \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n\} : P)/f}]}$$

R-ADAPT

$$\frac{}{\langle \Theta \rangle\ w \rightsquigarrow_{\mathrm{c}} w}$$

R-FREEZE-COMM

$$\frac{}{c\ \overline{R}\ \overline{w} \rightsquigarrow_{\mathrm{c}} \lceil c\ \overline{R}\ \overline{w} \rceil}$$

R-FREEZE-FRAME-USE

$$\frac{\neg(\mathcal{F}[\mathcal{E}]\ \text{handles}\ c)}{\mathcal{F}[\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil] \rightsquigarrow_{\mathrm{u}} \lceil \mathcal{F}[\mathcal{E}[c\ \overline{R}\ \overline{w}]] \rceil}$$

R-FREEZE-FRAME-CONS

$$\frac{\neg(\mathcal{F}[\mathcal{E}]\ \text{handles}\ c)}{\mathcal{F}[\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil] \rightsquigarrow_{\mathrm{c}} \lceil \mathcal{F}[\mathcal{E}[c\ \overline{R}\ \overline{w}]] \rceil}$$

R-LIFT-UU

$$\frac{m \rightsquigarrow_{\mathrm{u}} m'}{\mathcal{E}[m] \longrightarrow_{\mathrm{u}} \mathcal{E}[m']}$$

R-LIFT-UC

$$\frac{m \rightsquigarrow_{\mathrm{u}} m'}{\mathcal{E}[m] \longrightarrow_{\mathrm{c}} \mathcal{E}[m']}$$

R-LIFT-CU

$$\frac{n \rightsquigarrow_{\mathrm{c}} n'}{\mathcal{E}[n] \longrightarrow_{\mathrm{u}} \mathcal{E}[n']}$$

R-LIFT-CC

$$\frac{n \rightsquigarrow_{\mathrm{c}} n'}{\mathcal{E}[n] \longrightarrow_{\mathrm{c}} \mathcal{E}[n']}$$

Figure 3.4: Operational Semantics

mand c; i.e. it is a context of the form $u\ (\overline{t}, [\ ], \overline{u'})$ where u is a handler that handles c at the index corresponding to the hole. Thus, the whole term is frozen up to the first handler, at which point is it handled with R-HANDLE.

The R-LIFT rules then express that we can perform any of these reductions in any evaluation context.

Pattern Binding We now discuss the pattern binding rules of Figure 3.5.

$$\boxed{r : T \leftarrow t \dashv [\Sigma]\ \theta}$$

B-VALUE

$$\frac{\begin{array}{c}\Sigma \vdash \Delta \dashv \Sigma' \\ p : A \leftarrow w \dashv \theta\end{array}}{p : \langle\Delta\rangle A \leftarrow w \dashv [\Sigma]\ \theta}$$

B-REQUEST

$$\frac{\begin{array}{c}\Sigma \vdash \Delta \dashv \Sigma' \qquad \mathcal{E}\ \mathsf{poisedfor}\ c \\ \Delta = \Theta \mid \Xi \qquad c : \forall \overline{Z}.\overline{B \to}\, B' \in \Xi \qquad (p_i : B_i \leftarrow w_i \dashv \theta_i)_i\end{array}}{\langle c\ \overline{p} \to z\rangle : \langle\Delta\rangle A \leftarrow \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil \dashv [\Sigma]\ \overline{\theta}[\uparrow(\{x \mapsto \mathcal{E}[x]\} : \{B' \to [\Sigma']A\})/z]}$$

B-CATCHALL-VALUE

$$\frac{\Sigma \vdash \Delta \dashv \Sigma'}{\langle x\rangle : \langle\Delta\rangle A \leftarrow w \dashv [\Sigma]\ [\uparrow(\{w\} : \{[\Sigma']A\})/x]}$$

B-CATCHALL-REQUEST

$$\frac{\begin{array}{c}\Sigma \vdash \Delta \dashv \Sigma' \qquad \mathcal{E}\ \mathsf{poisedfor}\ c \\ \Delta = \Theta \mid \Xi \qquad c : \forall \overline{Z}.\overline{B \to}\, B' \in \Xi\end{array}}{\langle x\rangle : \langle\Delta\rangle A \leftarrow \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil \dashv [\Sigma]\ [\uparrow(\{\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil\} : \{[\Sigma']A\})/x]}$$

$$\boxed{p : A \leftarrow w \dashv \theta}$$

B-VAR

$$\frac{}{x : A \leftarrow w \dashv [\uparrow(w : A)/x]}$$

B-DATA

$$\frac{k\ \overline{A} \in D\ \overline{R} \qquad (p_i : A_i \leftarrow w_i \dashv \theta_i)_i}{k\ \overline{p} : D\ \overline{R} \leftarrow k\ \overline{w} \dashv \overline{\theta}}$$

Figure 3.5: Pattern Binding

Chapter 4

Pre-emptive Concurrency

4.1 Motivation

Our scheduler in Section 2.3 relies on threads yielding. This is fine for small examples, but when working with longer programs this is inconvenient; the programmer must insert yields with a consistent frequency, so as to avoid process starvation. It would be better to just use some automatic way of yielding.

Consider the two programs below;

```
controller : {[Stop, Go, Console] Unit}
controller! =
    stop!; print stop ; sleep 200000; go!; controller!

runner : {[Console] Unit}
runner! = print ``1 ``; print ``2 ``; print ``3 '';
```

We want a multihandler that uses the `stop` and `go` commands from `controller` to control the execution of `runner`. The console output of this multihandler should be then `1 stop 2 stop 3 stop`.

We can simulate this behaviour by using the familiar `Yield` interface from Section 2.3.1.

```
runner : {[Console, Yield] Unit}
runner! = print "1 "; yield!; print "2 "; yield!; print "3 "; yield!

suspend : {<Yield> Unit -> <Stop, Go> Unit
  -> Maybe {[Console, Yield] Unit} -> [Console] Unit}
suspend <yield -> r> <stop -> c> _ =
    suspend unit (c unit) (just {r unit})
suspend <_>          <go -> c>   (just res) =
```

```
    suspend res! (c unit) nothing
suspend unit          <_>          _ = unit
```

Running `suspend runner! controller! nothing` then prints out `1 stop 2 stop 3` as desired. This is due to the same synchronisation behaviour that we saw in Section 2.3.1; `runner` is evaluated until it becomes a command or a value, and then `controller` is given the same treatment. Once both are a command or a value, pattern matching is done.

This gives us the desired behaviour; the use of `yield` gives the controller a chance to run, and also gives us access to the continuation of `runner`. We can then use this to implement whatever scheduling strategy we like. We are, however, still operating co-operatively; the programmer has to manually insert `yield` commands. If the programmer does not do so evenly enough then either process could become starved. As such, we continue searching for a better solution.

4.2 Relaxing Catches

One approach is to relax the rules for pattern matching with the catchall pattern $\langle x\rangle$. This would let us match generic commands that may not be handled by the current handler. The key to implementing this lies in the pattern binding rules of Figure 3.5; specifically B-CATCHALL-REQUEST.

The crux is that the command c that is invoked in the frozen term $\lceil \mathcal{E}[c\,\overline{R}\,\overline{w}]\rceil$ must be a command offered by the extension Ξ; that is, it must be handled by the current use of R-HANDLE. Refer back to the example of Section **??**. This rules means that the catch-all pattern `<_>` in the final pattern matching case of `suspend` can match against `stop` or `go`, as they are present in the extension of the second argument, but not `print` commands; although the `Console` interface is present in the ability of `controller`, it is not in the extension in `suspend`.

B-CATCHALL-REQUEST-LOOSE

$$\frac{\Sigma \vdash \Delta \dashv \Sigma'}{\langle x\rangle : \langle\Delta\rangle A \leftarrow \lceil \mathcal{E}[c\,\overline{R}\,\overline{w}]\rceil \dashv_{[\Sigma]} [\uparrow(\{\lceil \mathcal{E}[c\,\overline{R}\,\overline{w}]\rceil\}:\{[\Sigma']A\})/x]}$$

Figure 4.1: Updated B-CATCHALL-REQUEST

In the interests of pre-emption, we propose to remove this constraint from B-

CATCHALL-REQUEST, replacing the rule with B-CATCHALL-REQUEST-LOOSE as seen in Figure 4.1. This lets us update the previous `suspend` code to the following, which yields the same results as last time;

```
runner : {[Console] Unit}
runner! = print "1 "; print "2 "; print "3 "

suspend : {Unit -> <Stop, Go> Unit -> Maybe {[Console] Unit} -> [
   Console] Unit}
suspend <r> <stop -> c> _ =
    suspend unit (c unit) (just r)
suspend <_>          <go -> c>   (just res) =
    suspend res! (c unit) nothing
suspend unit         <_>          _ = unit
```

Now when we run `suspend runner! controller! nothing`, the `suspend` handler can match the catchall pattern `<r>` against the `print` commands in `runner`.

The no-snooping policy with respect to effect handlers (Convent et al. [2020]) states that a handler should not be able to intercept effects that it does not handle. This change breaks this policy, as we can now tell when an command is used. Whilst we can not handle it as per usual, we get the option to throw away the continuation. A system that does not allow for snooping is much preferred.

4.3 Freezing Arbitrary Terms

The approach of Section 4.2 can only interrupt command invocations. If `runner` were instead a sequence of pure computations[1] we would be unable to interrupt it; it does not invoke commands.

As such, we need to further change the pattern binding rules of Figure 3.5. This is to let us interrupt arbitrary computation terms. In Figure 4.2, we see an updated version of the runtime syntax; this allows for the suspension of arbitrary *uses*.

Note that frozen terms here behave in a similar way to frozen commands, by freezing the rest of the term around it as well. This continues up until a handler is reached, at which point the term is unfrozen and resumed. This process of freezing up to a handler is enforced by the predicate $\mathcal{F}$ $\mathsf{not\ handler}$, which is true only when $\mathcal{F}$ is of the form $u\ (\bar{t}, [\,], \bar{n})$.

[1] I.e. `runner! = 1 + 1; 1 + 1; 1 + 1`; ...

$$
\begin{array}{lrl}
\text{(uses)} & m ::= & \ldots \mid \lceil \mathcal{E}[c\,\overline{R}\,\overline{w}] \rceil \mid \boxed{\lceil m \rceil} \\
\text{(constructions)} & n ::= & \ldots \mid \lceil \mathcal{E}[c\,\overline{R}\,\overline{w}] \rceil \mid \boxed{\lceil m \rceil} \\
\text{(use values)} & u ::= & x \mid f\,\overline{R} \mid \uparrow(v : A) \\
\text{(non-use values)} & v ::= & k\,\overline{w} \mid \{e\} \\
\text{(construction values)} & w ::= & \downarrow u \mid v \\
\text{(normal forms)} & t ::= & w \mid \lceil \mathcal{E}[c\,\overline{R}\,\overline{w}] \rceil \mid \boxed{\lceil m \rceil} \\
\text{(evaluation frames)} & \mathcal{F} ::= & [\,]\,\overline{n} \mid u\,(\overline{t}, [\,], \overline{n}) \mid \uparrow([\,] : A) \\
 & \mid & \downarrow[\,] \mid k\,(\overline{w}, [\,], \overline{n}) \mid c\,\overline{R}\,(\overline{w}, [\,], \overline{n}) \\
 & \mid & \mathbf{let}\ f : P = [\,]\ \mathbf{in}\ n \mid \langle \Theta \rangle\ [\,] \\
\text{(evaluation contexts)} & \mathcal{E} ::= & [\,] \mid \mathcal{F}[\mathcal{E}]
\end{array}
$$

Figure 4.2: Runtime Syntax, Updated with Freezing of Uses

$\boxed{m \rightsquigarrow_{\mathrm{u}} m'}$ $\boxed{n \rightsquigarrow_{\mathrm{c}} n'}$

$$
\textsc{R-Freeze-Use}\ \frac{}{m \rightsquigarrow_{\mathrm{u}} \lceil m \rceil}
\qquad
\textsc{R-Freeze-Frame-Use}\ \frac{\mathcal{F}\ \text{not handler}}{\mathcal{F}[\mathcal{E}[\lceil m \rceil]] \rightsquigarrow_{\mathrm{u}} \lceil \mathcal{F}[\mathcal{E}[m]] \rceil}
\qquad
\textsc{R-Freeze-Frame-Cons}\ \frac{\mathcal{F}\ \text{not handler}}{\mathcal{F}[\mathcal{E}[\lceil m \rceil]] \rightsquigarrow_{\mathrm{c}} \lceil \mathcal{F}[\mathcal{E}[m]] \rceil}
$$

Figure 4.3: Updated Freezing

With this in mind, we now give the updated rule for the catchall pattern matching on frozen terms. This can be seen in Figure 4.4. It expresses that an arbitrary frozen term can be matched against the computation pattern $\langle x \rangle$. The suspended, unfrozen computation $\{m\}$ is then bound to x, in a similar way to other B-CATCHALL rules. Observe that this maintains no-snooping; we don't know that the frozen computation performed an effect.

We can simply reuse the `suspend` handler from Section 4.2. Everything works largely the same; we run the leftmost argument until it freezes or invokes a command, at which point we run the next argument. The frozen term can then be bound to the catch-all pattern, if this is the pattern that matches.

4.4 Yielding

Observe that the freezing approach of Section 4.3 ends up reimplementing a lot of the behaviour of the freezing of ordinary commands, without adding much new behaviour.

$$\frac{\Sigma \vdash \Delta \dashv \Sigma'}{\langle x \rangle : \langle \Delta \rangle A \leftarrow \lceil m \rceil \dashv_{\Sigma} [\uparrow(\{m\}:\{[\Sigma']A\})/x]}\ \textsc{B-CatchAll-Freeze}$$

Figure 4.4: Catching Interrupts rule.

It turns out that we can get the exact same behaviour by just inserting a command invocation into the term instead, and handling this as normal.

Recall the simple Yield effect from Section 2.3; it supports one operation, yield : Unit. Whilst it sounds boring from the type, remember that the invocation of an effect offers up the continuation of the program as a first-class value, so that they might concurrently run the function with other functions or control execution by some other means.

Our solution is simple; whenever we are in an evaluation context where the ability contains the Yield effect, we insert an invocation of yield before the term in question. This is expressed formally in Figure 4.5. We refer to this system as $\mathbb{F}_{\mathcal{ND}}$.

TODO: Add rules for eval ctxs converting use to const, use to use, etc

$\boxed{n \leadsto_{\mathrm{u}} n'}$

$$\frac{\mathcal{E} \text{ allows yield}}{\mathcal{E}[n] \leadsto_{\mathrm{u}} \mathcal{E}[\mathsf{yield!}; n]}\ \textsc{R-Yield-EF}$$

Figure 4.5: Inserting Yields

Note that R-YIELD-EF relies on the predicate $\mathcal{E}$ allows c. For any frame apart from argument frames, $\mathcal{F}[\mathcal{E}]$ allows $c =$ false. In this case, it is defined as follows;

$$\uparrow(v : \{\overline{\langle \Delta \rangle A \to} [\Sigma]B\})\ (\bar{t}, [\,], \bar{n}) \text{ allows } c = \Xi_{|\bar{t}|} \text{ allows } c$$
$$\text{where } \Delta_{|\bar{t}|} = \Theta_{|\bar{t}|} \mid \Xi_{|\bar{t}|}$$

For an extension Ξ, the predicate Ξ allows c is true if $c \in I$ for some $I \in \Xi$.

Informally, $\mathcal{E}$ allows c is true when $\mathcal{E}$ is a handler, and the extension at the hole contains an interface which offers yield as a command. For instance, if a handler had type `{<Yield>X -> Y -> [Yield]X}`, the first argument would be allowed to yield but the second would not.

We also make use of an auxiliary combinator $\_;\_$. This is the traditional sequential composition operator snd $x\ y \mapsto y$, where both arguments are evaluated and the result of the second one is returned. We see that it would be a type error if we were to insert a `yield` command in a context where yield was not a part of the ability. In the context of R-YIELD-EF this means we will perform the yield operation and then the use m, but discard the result from yield.

Observe that this gives us fine-grained control over which parts of our program become asynchronous. One might want a short-running function to not be pre-emptible and just run without pause; conversely, one might want a long-running function to be interruptible. The programmer gets to choose this by labelling the functions with yield in the ability. This is one improvement over the system of Section 4.3, another being that we define fewer new rules and constructs.

Nondeterminism Observe that this system, and the system from Section 4.3, are both nondeterministic. This is because at any point we have the opportunity to either invoke yield (respectively freeze the term), or continue as before.

Consider running `hdl (print "A")(print "B")`, for some binary multihandler `hdl`. We could evaluate `print "A"` first and then `print "B"`, or freeze `print "A"` and evaluate `print "B"` first. Both of these would obviously result in different things printed to the console.

4.5 Counting

The semantics given by Section 4.4 is fine, but is non-deterministic; at any point, we can choose to either insert a yield invocation or carry on as normal. Furthermore, we do not particularly need to yield very frequently; we might rather yield every 1000 reduction steps or so.

As such, we supplement the operational semantics with a counter c_y. This counter has two states; it could either be counting up, which is the form $c(n)$ for some n, or it is a signal to yield as soon as possible, which is the form yield.

To increment this counter, we use a slightly modified version of addition, denoted $+_c$. This is simply defined as

$\boxed{m \rightsquigarrow_{\mathrm{u}} m'}$ $\boxed{n \rightsquigarrow_{\mathrm{c}} n'}$ $\boxed{m \longrightarrow_{\mathrm{u}} m'}$ $\boxed{n \longrightarrow_{\mathrm{c}} n'}$

R-HANDLE-COUNT

$$\frac{k = \min_i \{i \mid \exists\overline{\theta}.(r_{i,j} : \langle\Delta_j\rangle A_j \leftarrow t_j \dashv[\Sigma]\ \theta_j)_j\} \qquad (r_{k,j} : \langle\Delta_j\rangle A_j \leftarrow t_j \dashv[\Sigma]\ \theta_j)_j}{\uparrow(\{((r_{i,j})_j \to n_i)_i\} : \{\overline{\langle\Delta\rangle A \to}\ [\Sigma]B\})\ \bar{t}; c(n) \ \rightsquigarrow_{\mathrm{u}}\ \uparrow((\overline{\theta}(n_k) : B)); n +_c 1}$$

R-YIELD-CAN

$$\frac{\mathcal{E} \text{ allows yield}}{\mathcal{E}[m]; \mathsf{yield} \ \rightsquigarrow_{\mathrm{u}}\ \mathcal{E}[\mathsf{yield!}; m]; c(0)}$$

R-YIELD-CAN'T

$$\frac{\neg(\mathcal{E} \text{ allows yield}) \qquad m; c(n) \rightsquigarrow_{\mathrm{u}} m'; c'}{\mathcal{E}[m]; \mathsf{yield} \ \rightsquigarrow_{\mathrm{u}}\ \mathcal{E}[m]; \mathsf{yield}}$$

Figure 4.6: Yielding with Counting

$$x +_c y = \begin{cases} c(x+y) & \text{if } x+y \le t_y \\ \mathsf{yield} & \text{otherwise} \end{cases}$$

where t_y is the threshold at which we force a yield.

The transitions in our operational semantics now become of the form $m; c_y \rightsquigarrow_{\mathrm{u}} m'; c_y{}'$. In Figure 4.6 we give an updated rule for R-HANDLE — overwriting the previous rule — and two new rules for inserting yields. We refer to this system as $\mathbb{F}_C$.

R-HANDLE-COUNT replaces the previous rule R-HANDLE. If the counter is in the state $c(n)$, we perform the handling as usual, incrementing the counter by 1. Here we use $+_c$, which will set the counter to be yield if the addition brings it over the threshold value.

R-YIELD-CAN and R-YIELD-CAN'T dictate what to do if we have to yield as soon as possible. If the evaluation context allows yield commands to be inserted we do so and reset the counter. If not, but the term could otherwise reduce if the counter had a different value, then we make that transition, still maintaining the yield signal.

Dolan et al. take a similar approach to this when investigating asynchrony in Multicore OCaml (Dolan et al. [2017]). They rely on the operating system to provide a timer interrupt, which is handled as a first-class effect. Our system is more self-contained; the timing is implemented within the language itself and doesn't rely on the operating system providing interrupts. Furthermore, we get fine-grained control over when the timer can fire, as we can choose to put yield in the ability of interruptible terms.

Determinism Observe that the semantics of Frank equipped with the rules in Figure **??** are now deterministic; for any term and counter pair, there is only one possible reduction we can make. This is a clear improvement on the nondeterministic semantics of Section 4.4, whilst still maintaining a similar behaviour; we are simply *restricting* the parts of the program where we can yield.

We can characterise this by saying that $\mathbb{F}_C$ *implements* $\mathbb{F}_{\mathcal{ND}}$; that is to say the counting system gives a deterministic way to perform the nondeterministic system.

Theorem 1 (Counting Implements Nondeterminism)

- *For any use m and counter c_y, if $m, c_y \leadsto_{\mathrm{u}} m', c_y'$ in $\mathbb{F}_C$ then $m \leadsto_{\mathrm{u}} m'$ in $\mathbb{F}_{\mathcal{ND}}$.*
- *For any construction n and counter c_y, if $n, c_y \leadsto_{\mathrm{u}} n', c_y'$ in $\mathbb{F}_C$ then $n \leadsto_{\mathrm{u}} n'$ in $\mathbb{F}_{\mathcal{ND}}$.*

One might consider a different approach, rather than a global counter, which would also implement the nondeterministic semantics.

4.6 Handling

Observe that we can now use the same `suspend` handler from Section **??**, without having to manually insert yield commands in `runner`. Assuming the threshold t_y is set to 1, the following code will give the desired output.

```
runner : {[Console] Unit}
runner! = print "1 "; print "2 "; print "3 "

suspend : {<Yield> Unit -> <Stop, Go> Unit
  -> Maybe {[Console, Yield] Unit} -> [Console] Unit}
suspend <yield -> r> <stop -> c> _ =
    suspend unit (c unit) (just {r unit})
suspend <_>           <go -> c>    (just res) =
    suspend res! (c unit) nothing
suspend <yield -> r> <c>           = suspend (r unit) c!
suspend unit         <_>          _ = unit

  suspend <yield -> k> a = blah
```

The first argument is evaluated until the counter is greater than the threshold, at which point a yield command is performed; the rest of the computation is then frozen and the second argument is evaluated. Observe that the Yield interface is not present in the adjustment of the second argument, so it is left to run as normal.

We might also want to make the controller — being the second argument — pre-emptible; it might do some other long-running computation in between performing `stop` and `go` operations. We have to add Yield to the adjustment at the second argument, but also more pattern matching cases.

```
suspend : {<Yield> Unit -> <Stop, Go, Yield> Unit
  -> Maybe {[Console, Yield] Unit} -> [Console] Unit}
suspend <yield -> r> <stop -> c> _ =
    suspend unit (c unit) (just {r unit})
suspend <_>          <go -> c>    (just res) =
    suspend res! (c unit) nothing
suspend <yield -> r> <yield -> c> = suspend (r unit) (c unit)
suspend <yield -> r> <c> = suspend (r unit) c!
suspend <r> <yield -> c> = suspend r!  (c unit)
suspend unit         <_>          _ = unit
```

These let yield commands synchronise with each other, achieving fair scheduling, as discussed in Section 2.3. It is a bit annoying to write these by hand, as they take up a lot of space and are orthogonal to the rest of the logic of the handler.

It is fortunate then that this process of resuming as many yields as possible can be automated completely. Given a multihandler with m arguments, n of which have Yield in their adjustment, we first try and resume all n yield commands. After this we try and resume all of the different permutations of $n-1$ yield commands, and so on until we are trying to resume 0 yield commands. The resuming clauses for the 3-ary case can be seen below.

```
sch3 : {<Yield> Unit -> <Yield> Unit -> <Yield> Unit -> Unit}
sch3 <yield -> h> <yield -> j> <yield -> k> =
    sch3 (h unit) (j unit) (k unit)

sch3 <yield -> h> <yield -> j> <k> = sch3 (h unit) (j unit) k!
sch3 <yield -> h> <j> <yield -> k> = sch3 (h unit) j! (k unit)
sch3 <h> <yield -> j> <yield -> k> = sch3 h! (j unit) (k unit)

sch3 <yield -> h> <j> <k> = sch3 (h unit) j! k!
sch3 <h> <yield -> j> <k> = sch3 h! (j unit) k!
sch3 <h> <j> <yield -> k> = sch3 h! j! (k unit)
```

These commands can then be inserted generically at runtime. If no other hand-written patterns match, we insert these patterns and try all of these. It is important to try these after the rest of the patterns, as the use may want to handle yield commands some other way; we do not want to interfere with this. This means we can program in a direct manner manner, easily toggling which arguments should be interruptible by adding Yield to the corresponding interface.

Automatically inserting yield-handling clauses when combined with automatically *inserting* yield commands then gives us pre-emptive concurrency at no overhead to the programmer.

4.7 Starvation

Consider the following program;

```
echo : {String -> [Console, Yield] Unit}
echo st = print st; echo st

sched : {<Yield> Unit -> <Yield> Unit -> Unit}
sched unit unit = unit

tree : {[Console] Unit}
tree! = sched (echo "A ")
              (sched (echo "B ") (echo "C "))
```

We would like `tree!` to print out `"A B C A B C A B C ..."`. However, when using the insertion and handling of yield commands as discussed, the result is `"A B C B C B C ..."`. The `echo "A "` thread is *starved* of processor time. This happens because when `echo "B "` yields the command is immediately handled by the lower `sched` handler and `echo "C "` is ran (and vice versa). Ideally what we want is to break out of this cycle and yield on top of the lower scheduler.

To do this we need to maintain a series of counters; one for each multihandler. If we consider the `tree` as a tree where `echo` functions are at the leaves and `sched` makes up the branches, we also can now yield at the branches as well as the leaves.

We have to update the semantics with the following rules to make this possible. Our transitions are now of the form $m, \overline{c_y} \leadsto_{\mathrm{u}} m', \overline{c_y}$. This list of counters is the list of counters *above* the current term in this tree of multihandlers. For instance, `echo "A "` will maintain the counter for `sched` directly above it; but `echo "B "` maintains the counter for both multihandlers above. The rules are shown in Figure 4.7.

ADD-COUNTER

$$\frac{\mathcal{E}[n]; c_n, \ldots, c_1 \rightsquigarrow_{\mathrm{u}} \mathcal{E}[n']; c_n, \ldots, c_1 \qquad \mathcal{F} \text{ is handler}}{\mathcal{F}[\mathcal{E}[n]]; c_{n+1}, c_n, \ldots, c_1 \rightsquigarrow_{\mathrm{u}} \mathcal{F}[\mathcal{E}[n']]; c_{n+1}, c_n, \ldots, c_1}$$

R-HANDLE

$$\frac{\begin{array}{c} k = \min_i \{ i \mid \exists \overline{\theta}.(r_{i,j} : \langle \Delta_j \rangle A_j \leftarrow t_j \dashv_{[\Sigma]} \theta_j)_j \} \\ (r_{k,j} : \langle \Delta_j \rangle A_j \leftarrow t_j \dashv_{[\Sigma]} \theta_j)_j \qquad \forall\, j \leq n \,.\, c_j \neq \text{yield} \end{array}}{\mathcal{E}[\uparrow(\{((r_{i,j})_j \rightarrow n_i)_i\} : \{\overline{\langle \Delta \rangle A \rightarrow}\ [\Sigma] B\})\ \overline{t}]; c_n, \ldots, c_1 \rightsquigarrow_{\mathrm{u}} \mathcal{E}[\uparrow((\overline{\theta}(n_k) : B))]; c_n, \ldots, c_1}$$

R-YIELD-CAN

$$\frac{\mathcal{E} \text{ allows Yield} \qquad \forall\, j \leq (n-1) \,.\, c_j \neq \text{yield}}{\mathcal{E}[m]; c_n, \ldots, c_2, \text{yield} \rightsquigarrow_{\mathrm{u}} \mathcal{E}[\text{yield!}; m]; c_n, \ldots, c_2, c(0)}$$

R-YIELD-CAN'T

$$\frac{\begin{array}{c} \neg(\mathcal{E} \text{ allows Yield}) \\ \forall\, j \leq (n-1) \,.\, c_j \neq \text{yield} \qquad \mathcal{E}[m]; c_n, \ldots, c_2, c(k) \rightsquigarrow_{\mathrm{u}} \mathcal{E}[m']; c_n, \ldots, c_2, c_1' \end{array}}{\mathcal{E}[m]; c_n, \ldots, c_2, \text{yield} \rightsquigarrow_{\mathrm{u}} \mathcal{E}[m']; c_n, \ldots, c_2, \text{yield}}$$

Figure 4.7: New counting, backwards ordered though

R-YIELD-CAN and R-YIELD-CAN'T are fairly similar to the rules in Figure 4.6. The main difference is that we can only insert a yield if every counter above ours is not also trying to yield. This is important as it gives priority to multihandlers higher up in our syntax tree.

R-HANDLE is also fairly similar to previous versions of the rule. The difference again is that we don't allow a transition if any of the earlier counters are in the yieldposition; they must yield before we can progress. Once we do handle the operation we increase all of the counters in the stack.

TODO: Add counter????

4.8 Soundness

We now state the soundness property for our extended system, as well as the subject reduction theorem needed for this proof. Our system is nothing more than the system of Convent et al. with extra rules; as such we omit most of the details.

Theorem 2 (Subject Reduction)

- *If* $\Phi;\Gamma\,[\Sigma]\!\vdash m \Rightarrow A$ *and* $m; c_y \rightsquigarrow_{\mathrm{u}} m'; c_y{}'$ *then* $\Phi;\Gamma\,[\Sigma]\!\vdash m' \Rightarrow A$.
- *If* $\Phi;\Gamma\,[\Sigma]\!\vdash n : A$ *and* $n; c_y \rightsquigarrow_{\mathrm{c}} n'; c_y{}'$ *then* $\Phi;\Gamma\,[\Sigma]\!\vdash n' : A$.

The proof follows by induction on the transitions $\rightsquigarrow_{\mathrm{u}}, \rightsquigarrow_{\mathrm{c}}$. We first consider the two possible states for c_y. If it is in the form $c(n)$, then the reduction rules are simply the same as in Convent et al. [2020], as we do not change the counter. The only exception to this is the updated R-HANDLE rule, which is the same as before except for modifications to the counter; regardless of the counter, the resulting term m' still remains the same type.

Thus the only new cases are R-YIELD-CAN and R-YIELD-CAN'T.

Case R-YIELD-CAN By the assumption we have that $\mathcal{E}$ allows yield. This only holds if the context is of the form

$$\mathcal{E}[\,] = \uparrow(v : \{\overline{\langle\Delta\rangle A \rightarrow [\Sigma]B}\})\ (\bar{t}, [\,], \overline{n'})$$

Assume that

$$\Phi;\Gamma\,[\Sigma]\!\vdash \uparrow(v : \{\overline{\langle\Delta\rangle A \rightarrow [\Sigma]B}\})\ (\bar{t}, \mathcal{E}'[n], \overline{n'}) \Rightarrow B$$

From $\mathcal{E}$ allows yield we know that yield $\in \Xi_{|\bar{t}|}$ where $\Delta_{|\bar{t}|} = \Theta_{|\bar{t}|} \mid \Xi_{|\bar{t}|}$. Then by inversion on T-APP we have $\Phi;\Gamma\,[\Sigma'_{|\bar{t}|}]\!\!- \mathcal{E}'[n] : A_{|\bar{t}|}$ and $\Sigma \vdash \Delta_{|\bar{t}|} \dashv \Sigma_{|\bar{t}|}$. It follows than that $\Phi;\Gamma\,[\Sigma'_{|\bar{t}|}]\!\!- \mathcal{E}'[\mathsf{yield}; n] : A_{|\bar{t}|}$, as we know that yield commands are permitted under ability $\Sigma'_{|\bar{t}|}$.

Case R-YIELD-CAN'T This case is more straightforward. By the assumption we have that the evaluation frame $\mathcal{F}$ does not permit yielding, but the term inside the frame could otherwise reduce.

Assume $\Phi;\Gamma\,[\Sigma]\!\!- \mathcal{F}[n] : A$, and therefore $\Phi;\Gamma\,[\Sigma]\!\!- n : A'$. By the assumption and subject reduction, $\Phi;\Gamma\,[\Sigma]\!\!- n' : A'$. Then clearly $\Phi;\Gamma\,[\Sigma]\!\!- \mathcal{F}[n'] : A$.

Theorem 3 (Type Soundness)

- *If $\cdot;\cdot\,[\Sigma]\!\!- m \Rightarrow A$ then either m is a normal form such that m respects Σ or there exists a unique $\cdot;\cdot\,[\Sigma]\!\!- m' \Rightarrow A$ such that $m \longrightarrow_{\mathrm{u}} m'$.*
- *If $\cdot;\cdot\,[\Sigma]\!\!- n : A$ then either n is a normal form such that n respects Σ or there exists a unique $\cdot;\cdot\,[\Sigma]\!\!- n' : A$ such that $n \longrightarrow_{\mathrm{c}} n'$.*

The proof proceeds by simultaneous induction on $\cdot;\cdot\,[\Sigma]\!\!- m \Rightarrow A$ and $\cdot;\cdot\,[\Sigma]\!\!- n : A$.

TODO: Talk about Soundness for $\mathbb{F}_{\mathcal{ND}}$ as well as $\mathbb{F}_{\mathcal{C}}$.

Chapter 5

Implementation

We now introduce the Frank library used for asynchronous effects. Our design closely follows the design of Æff (Ahman and Pretnar [2020]), a language designed around writing multithreaded programs that communicate by sending *interrupts*. A thread dictates how it will respond to an interrupt by installing an *interrupt handler*, also known as a *promise*. Interrupts and interrupt handlers can be seen as a less expressive version of effects and effect handlers; an interrupt handler describes how to behave on receipt of an interrupt, but it does not get access to the continuation of the caller.

Interrupts and interrupt handlers have one particularly compelling feature; when we invoke an interrupt (in the case of synchronous effects, this is just invoking a command), we can carry on computing the rest of the code whilst we wait for a response. This is a stark difference to normal effects, where the rest of the computation is blocked whilst we wait for an answer. The programmer can then choose to await the response from interrupt, which blocks computation until the interrupt handler has been fulfilled.

Unlike Æff, our system does not track the uses of asychronous effects. It does, however, track the traditional effects that are permitted in promises.

Æff's interrupt handlers are manifested in Frank through the **`Promise`** interface.

```
interface Promise =
      promise R : Prom R [Promise, RefState, Yield]
               -> Pid R  [Promise, RefState, Yield]
    | signal : Sig -> Unit
    | await R : Pid R [Promise, RefState, Yield] -> R

data Prom R [E] = prom {Sig -> Maybe {[E|]R}}
```

```
data Pid X = pid (Ref (PromiseStatus X))

data PromiseStatus X = waiting | done X | resume {X -> Unit}
```

The `promise R` command is a polymorphic command which takes a function of type `Sig -> Maybe {[E|] R}`. This function is an interrupt handler; it dictates what to do on receipt of an interrupt, which is a thing of type `Sig`. The return type of the interrupt handler is `Maybe {[E|] R}`; this is because the programmer has the chance to return `nothing`, which will mean the promise goes unfulfilled and waits for another message. The programmer would want to do this on receipt of other types of message, or if a certain condition regarding the interrupt is not fulfilled[1]. An interrupt handler corresponding to an asynchronous division could be;

```
div : {Sig -> Maybe {[Promise] Unit}}
div (ask n d) = if (n > 0)
                   {just {signal (response (n / d))}}
                   {nohing}
div _ = nothing
```

We call the thunk `signal (response (n / d))` the *body* of the interrupt handler. We say that an interrupt handler *fires* if an interrupt is received that results in a non-`nothing` value. Observe that whilst we can perform some computation when deciding whether or not to fire based on an interrupt, the type of `Prom R` restricts us so that we cannot perform any effects whilst deciding; we may only perform effects in the body of the promise.

Once handled, `promise R` operation returns a `Pid R`. This contains information about the status of the promise; it is either empty, to signify the promise is unfulfilled, or it holds the result of the promise, or it holds a resumption that is automatically invoked when the promise is finished. Importantly, a function installing a promise should not have access to the body of this; it would ideally be some abstract type. The calling function should only interact with the `Pid` through the `await` command.

`signal` is a more simple operation. The `Sig` data type is the type of signals that the thread can invoke. These can hold extra data, also called a *payload*. For instance, if we had a program running remote function calls, we might have `Sig = call ArgsType | result ResultType`; the payloads are respectively of types `ArgsType` and `ResultType`. The handler for `Promise` will then send the signals to each other thread, possibly executing the interrupt handler if needs be.

[1]Interrupt handlers which put conditions on the incoming interrupts are called *guarded* interrupt handlers.

`await` takes a `Pid R` and returns a value of type `R`. This `R` is the returned value of the promise. `await` will block until the promise it awaits has been fulfilled; we come on to how it does so later.

Effect Typing We can track and control the effects that promises can perform using Frank's effect type system. Recall that Frank effect types implicitly add effect type variables to effect type declarations (as discussed in Section 2.2); thus the type `[ Promise, RefState, Yield]` desugars to `[E | Promise [E|], RefState, Yield ]`. Thus a Frank function of type `[Promise [Console]] X` can install promises that use the effects in the interface `[Promise [Console], Console, RefState, Yield ]`. We use a recursive type so that promises can themselves install other promises. `RefState` and `Yield` are explicitly added to the ability as a convenience, as every promise requires it in their ability for reasons that become clear later.

Threads To run several threads in parallel, we need to maintain a collection of thread states. When we stop executing a thread we suspend it, storing the continuation, and start executing a new one, in the same style as Section 2.3. We also need to store the promises that each thread has installed. These are stored as a stack to maintain the order of installation. Installing a promise is as straightforward as pushing it onto the corresponding thread's promise stack.

```
data Threads =
    tentry Int {[RefState, Yield] Unit}
           (TStack {Sig -> {[RefState, Yield] Unit}
                        -> Maybe {[RefState, Yield] Unit}})
           Threads
    | tnil
```

This collection is realised in Frank as the `Threads` datatype. It is essentially just a list of three-tuples[2], storing the integer ID, suspended computation, and promise stack of each thread.

Observe that installed promises take two arguments, instead of just one. The new argument is the suspended computation thus far. This is because we often have to take the body of the promises, compose it with the rest of the interrupted computation and then rehandle it with the `Promise` handler. If the promise results in `Nothing`, we don't remove it from the stack and it remains installed; recall that returning `Nothing`

[2]If Frank had support for type aliases, this is exactly what it would be.

signifies the promise not firing. If it returns `Just th`, we replace the currently stored suspended computation with `th`. We go into further detail about this later.

5.1 Handling Promises

We now introduce the handler for `Promise` effects.

```
hdl : {Int -> Ref Threads
    -> <Promise> Unit
    -> [RefState, Yield] Unit}
```

The first argument is the id of the thread being handled. The second one is a reference to the threads structure. These are parametrised by the effects performed in the promises, just like the `Promise` interface. The third argument is the thread itself, which performs `Promise` operations. Finally, the return type expresses that this code can perform any of the effects that the promises perform, plus `RefState` effects; the `Yield` encodes that this can be interrupted (as in Chapter 4).

Promises

```
1  hdl thId thrs <promise (prom cb) -> k> =
2      let cell = pid (new waiting) in
3      let cbMod = (toWrite cell cb) in
4      let cbMaybe = {sig rest -> case (cbMod sig)
5            { nothing -> nothing
6            | (just susp) ->
7                  just { hdl thId thrs (susp!; <Promise> rest!)} }} in
8      let queued = (addCb thId cbMaybe (read thrs)) in
9      write thrs queued;
10     hdl thId thrs (k cell)
```

Above we see the handler for `promise`. Line 2 creates a new reference cell for the promise id; this is initialised to `waiting`, as nothing has been performed yet. Line 3 calls the utility function `toWrite`, shown below.

```
toWrite : Pid S R [RefState]
          -> {S -> (Maybe {[RefState] R})}
          -> {S -> (Maybe {[RefState] Unit})}
toWrite (pid cell) cb =
    {x -> case (cb x)
         { nothing -> nothing
         | (just susp) ->
```

```
                just {case (read cell)
                        { empty -> write cell (done susp!)
                        | (resume resumption) -> resumption susp!}} }}
```

This takes a promise of type `Sig -> Maybe {R}` and converts it to type `Sig -> Maybe {Unit}`. `toWrite` modifies the given callback so once it has been executed it looks inside the given `Pid` cell. If the cell is `empty`, we just write the return value of the promise body in the cell. If there is a resumption, we resume it with the value of the promise body.

In lines 4-7 we convert the promise to also take the computation it interrupts as an argument. The body of the promise is composed with the interrupted computation and the whole thing is rehandled by the promise handler. This is essential to get blocking when installed from a promise working. For instance, we might have a program like `thr`;

```
thr : {[Promise] Unit}
thr! = promise (prom {stop -> await (promise (prom {go -> unit}))});
       otherComputation!
```

The desired behaviour is to start blocking when a `stop` message comes in, and then start running again once a `go` message is received. If we were just to handle rehandle the promise body separate from the interrupted computation, `otherComputation` would not get blocked.

Finally in lines 8-10 we add the new promise to the stack of promises installed for this thread and resume the computation with the cell we created earlier.

Signals

```
hdl thId thrs <signal sig -> thr> =
    let newThrs = runThreads sig (read thrs) in
    write thrs newThrs;
    hdl thId thrs (thr unit)
```

When a thread we're running invokes a signal, we inform the other threads of this using `runThreads`. This function then calls the `runThread` function on every thread;

```
runThread sig (trio susp cbs skipped) =
  case (dequeue cbs)
    { nothing -> trio susp cbs skipped
    | (just (pair cb cbs)) ->
      case (cb sig susp)
```

```
        { nothing -> runThread sig (trio susp cbs (enqueue cb
            skipped))
        | (just res) -> runThread sig (trio res cbs skipped)}}
```

We first check to see if there are any installed promises remaining. If there is, we run the promise with the signal supplied. We supply the callback with the incoming signal and the thunked computation thus far. If the callback returns **nothing** we reinstall it. If the callback gives us an updated thunk, we continue to run the rest of the promises, with this thunk the new computation to be extended.

At no point in this process are these thunks ever actually invoked; we are simply mutating suspended computations. When a signal triggers a promise, the body of the promise does not get performed until the thread is run by the scheduler. In Æff, when a signal is received by an interrupt handler there is some nondeterminism present; we can either trigger the interrupt handler or continue computing underneath the handler. In our system this choice is not available; the interrupt handler is always triggered first, and we process the body of it immediately.

Once promise execution is finished we update the state of **thrs** and resume handling, restarting the continuation with **unit** immediately. This is unlike traditional effect invocations, where we would block until a result is produced.

Await

```
hdl thId thrs <await cell -> thr> =
    case (readPid cell)
        { (done x) ->
            hdl thId thrs (thr x)
        | waiting ->
            writePid cell (resume thr);
            hdl thId thrs unit }
```

Handling **await** is surprisingly the simplest of the lot. Recall that **await** takes a promise id cell **Pid R** and returns a value of type **R**. The handler looks inside this cell; if there is a finished value there already (**done x**) it resumes the continuation with this value straight away. If the promise has not yet completed, we then write the resumption (which is of type **{R -> Unit}**) to the cell. The function **toWrite** used when installing promises changes the original promise to resume the continuation stored in **Pid**, if there is one present.

5.2 Multithreading

Multithreading then fits into our system in a natural way, via the `schedule` handler.

```
1 schedule : {<Yield> Unit -> Int -> Ref Threads -> [RefState]Unit}
2 schedule <yield -> k> cur thrs =
3     let next = nextId cur (keys (read thrs)) in
4     let newThk = lookupThk next (read thrs) in
5     let newThrs = writeThk cur {k unit} (read thrs) in
6     write thrs newThrs;
7     schedule newThk! next thrs
8
9 schedule unit cur thrs = scheduleT yield! cur thrs
```

Recall that the threads are stored with a thread id, an integer. We use these in our simple scheduling strategy, where we just cycle through all ids in ascending order. Line 3 finds the id of the next thread as per this strategy, and line 4 looks up the thunk from `Threads`. Line 5 then writes the current thread's thunk to `Threads`. Line 6 writes the updated version of `Threads` and line 7 starts executing the next continuation. Line 9 states that if a thread's value is unit we just force a yield. This is useful if a thread is blocked as we will instantly stop processing it and start the next one.

Chapter 6

Examples

The combination of our `Promise` interface with the pre-emptive concurrency of Section 4 lets us express complex concurrent programs in a simple, direct manner. In this section we show some examples of these.

TODO: Improve above paragraph

6.1 Pre-emptive Scheduling

Whilst we have already shown how to pre-emptively schedule several threads in Section 4.6, we might want to have a more robust way of doing this; the multihandler strategy is fixed in a left-to-right evaluation order.

TODO: Add more

```
data Sig = ... | stop Int | go Int
```

We add two more signals to the `Sig` datatype. The integer payload can act as a counter, or as a way to tell specific threads to stop or go. The blocking or non-blocking behaviour then depends on the promises for these signals.

```
onStop : {Int -> [Promise, Yield, RefState] Unit}
onStop id = let gp = promise (prom {s -> goPromise id s}) in
            await gp;
            promise (prom {s -> stopPromise id s});
            unit

stopPromise : {Int -> Sig -> Maybe {[Promise, Yield, RefState]Unit}}
stopPromise id (stop n) = if (n == id)
                             { just { onStop id } }
                             { nothing }
```

```
stopPromise id _ = nothing
```

`stopPromise` is another guarded interrupt handler; it will only fire its body if the payload to `stop` is the thread's ID. The body then installs a promise waiting for `go` and immediately awaits it, starting to block. The rest of the computation can not proceed until the corresponding go message is received. Once the correct go promise is received the promise is fulfilled adn the rest of the computation can start again, and the `stop` interrupt handler is reinstalled.

```
goPromise : {Int -> Sig -> Maybe {[Promise, Yield, RefState]Unit}}
goPromise id (go n) = if (n == id)
                         { just {unit} }
                         { nothing }
goPromise id _ = nothing
```

`goPromise` is simple in comparison; if it receives the correct `go` signal it just returns `unit`.

We can then install the stop interrupt handler before a computation to make it schedulable. The computation being scheduled is then unaware it is being scheduled, as desired.

```
counter : {Int -> [Console, Yield]Unit}
counter x = ouint x; print " "; sleep 200000; counter (x + 1)

thread : {[Promise, Console, RefState, Yield] Unit}
thread! = promise (prom {s -> stopPromise 0 s}); counter 0
```

Now all that remains is to have a source of `stop` and `go` signals. This could just be a standard round-robin scheduler or some more sophisticated strategy. One of the strengths of effect handlers for concurrency is the ability to abstract over scheduling strategies, and this strength is still present here.

6.2 Futures

Our developed asynchronous effects system is expressive enough to implement the asychronous post-processing of results, or *futures*, on top of what we already have. Previously these have had to be implemented as a separate language feature.

TODO: Reference for being a separate feature!

Futures are useful if we want to asynchronously perform some action once another promise has been completed. In the context of a web application, this might be updating the application's display once some remote call for data has finished. Observe that this differs from just awaiting the remote call and then updating once we have this; we do not want to block everything else from running, but want to perform this action asynchronously, when the promise is complete.

```
futureList : {Pid R [E|] -> {R -> [E|] Z} -> Sig -> Maybe {[E|] Z}}
futureList p comp (listSig _) =
    just { let res = await p in comp res}
futureList _ _ _ = nothing
```

When calling `futureList` we supply a promise of result type `R` and a computation of type `R -> Z`. We then await the promise, and once we have a value (of type `R`) run the computation with this. An example computation using this system is;

```
let recv = promise { (listSig xs) -> just {xs} | _ -> nothing} in
let prod = promise {s -> futureList filt product s} in
promise {s -> futureList prod {x -> signal (resultSig x)} s}
```

Where we, upon receipt of a list signal, take the product of the list element-wise and send another signal with this result. All three of these promises are triggered by the same signal; `recv` is executed first, which then executes `prod`, which then lets the final one run. This behaviour depends on signals being able to execute many promises at once (that is, behaving like *deep* rather than shallow handlers).

6.3 Async-Await with Workers

Our asynchronous effects system can express the familiar async-await abstraction. This had previously been implemented in Frank by forking new threads, then to be handled by a co-operative scheduler like in Section 2.3. We realise it by using a controller thread, which will send tasks to one of a set of n worker threads. When these worker threads are not working, they are instantly skipped; hence there is not much inefficiency associated with having extra idle workers.

We first show how the caller communicates with the controller.

```
resultWaiter : {Int -> Sig ->  Pid String [WebEffs]}
resultWaiter callNo (result res callNo') =
    if (callNo == callNo') { just {res} } { nothing }
resultWaiter _ _ = nothing

async : {[InTask] String} -> Ref Int -> Pid String [WebEffs]
async proc callCounter =
```

```
    let callNo = read callCounter in
    let waiter = {s -> resultWaiter callNo s} in
    signal (call proc callNo);
    write callCounter (callNo + 1);
    waiter
```

TODO: Talk about interface aliases

We use this function to issue a new asynchronous task. We keep a global counter to give each call a unique identifier. We then install an interrupt handler that waits for a result and simply returns it. Finally we send a `call` signal with the process and return the result interrupt handler. Observe that `async` returns the promise waiting for the correct result. As such, the `await` operation is just the built-in `await` operation; we don't need to define any extra functions.

`call` signals are handled by the controller thread. This thread keeps track of which of the workers do not currently have a task running, and also installs a promise to set its state to idle once the corresponding `result` message is received.

```
onResult : {Ref (List (Maybe Int)) -> Int -> Sig -> Maybe {[WebEffs]
    Unit}}
onResult active wid (result res cid) =
    just { write active (putIn nothing wid (read active)) }
onResult _ _ _ = nothing

onAsyncBody : {Ref (List (Maybe Int)) -> {[InTask] String} -> Int ->
    [WebEffs] Unit}
onAsyncBody active p callId =
    case (findFree (read active))
        { nothing -> print "All workers are busy."
        | (just wid) ->
            write active (putIn (just callId) wid (read active));
            promise (prom {s -> onResult active wid s});
            signal (workIn p wid callId)};
    promise (prom {s -> onAsync active s}); unit
```

Workers listen for a `workIn` message; when one comes in with their ID in the payload, they simply perform the computation and send a signal with the result.

```
workProm : {Int -> Sig -> Maybe {[WebEffs] Unit}}
workProm wid (workIn p wid' callId) =
    if (wid == wid')
       { just {let res = p! in
                signal (result res callId);
```

```
               worker wid; unit}}
       { nothing }
workProm _ _ = nothing

worker : {Int -> Pid Unit [WebEffs]}
worker wid = promise (prom {s -> workProm wid s})
```

This **result** signal triggers the interrupt handler installed by the **async** caller, but also triggers the promise installed by the controller, to inform it that the worker is now idle. This ability to trigger multiple promises with one message is a subtle but useful feature of the system.

6.4 Cancelling Tasks

Because we are working in a language equipped with effect handlers, we can easily write a handler for the Cancel effect, which just gets rid of the continuation and replaces it with some default value (e.g. **unit**).

```
interface Cancel = cancel : Unit

hdlCancel : {<Cancel> Unit -> Unit}
hdlCancel <cancel -> _> = unit
hdlCancel unit          = unit
```

WE can use this to cancel a task issued with **async**. Recall that tasks run on their own worker thread. As such, all we need to do to cancel them is reinstall the worker promise — which waits for new tasks — and then invoke the cancel effect. As such, our worker code becomes

```
canceller : {Int -> Int -> Sig -> Maybe {[WebEffs] Unit}}
canceller wid callId (cancelCall callId') =
    if (callId == callId')
        { just {worker wid; cancel!} }
        { nothing }
canceller _ _ _ = nothing

workProm : {Int -> Sig -> Maybe {[WebEffs] Unit}}
workProm wid (workIn p wid' callId) =
    if (wid == wid')
       { just { promise (prom {s -> canceller wid callId s} ;
               let res = p! in
               signal (result res callId);
```

```
              worker wid; unit}}
       { nothing }
workProm _ _ = nothing
```

We have to modify the handler for the Promise effect for this. Recall that when we install a promise, we convert it to a form that takes composes the body of the promise with the interrupted computation and rehandles the `Promise` effects that this might perform. We need to then wrap this in a handler for Cancel effects again; this is because user-level promises could perform Cancel effects, and we want to cancel the *whole* computation, not just the body of the promise.

```
case (cbMod sig)
    { nothing -> nothing
    | (just susp) ->
      just { stopCancel (hdl thId thrs (susp!; <LCancel, Promise>
         rest!)) }}
```

The realisation of cancellable function calls in Æff (Ahman and Pretnar [2020]) was to start awaiting a new promise that will never be installed. This leads to a space leak as unfulfilled promises build up. Our approach improves on this as the cancelled calls do genuinely disappear.

However, a weakness of ours is that we have to modify the handler code for promises, even though cancellation of calls and promises should be orthogonal.

6.5 Interleaving

With the Cancel effect, we can also define the useful `interleave` combinator, in the spirit of Koka's interleave operator Leijen [2017].

TODO: Think of a better way to make reference to Daan's work

```
resultWaiter : {String -> Int -> Int -> Int
    -> Maybe {[WebEffs] String}}
resultWaiter res callNo callA callB =
    if (callNo == callA)
        { just { signal (cancelCall callB); res} }
        { if (callNo == callB)
            { just { signal (cancelCall callA); res} }
            { nothing }}

interleave : {{[InTask] String} -> {[InTask] String}
          -> Ref Int -> [WebEffs] Pid String [WebEffs]}
```

```
interleave procA procB callCounter =
    let callA = read callCounter in
    write callCounter (callA + 1);
    let callB = read callCounter in
    write callCounter (callB + 1);

    let ileaveWaiter =
        promise (prom {(result res callNo) ->
            resultWaiter res callNo callA callB
                      | _ -> nothing
        }) in

    signal (call procA callA);
    signal (call procB callB);

    ileaveWaiter
```

This will issue the two tasks on different threads. We then install an interrupt handler for **`result`** signals; whichever result is received first we return and cancel the other task.

This lets us write timeouts for functions, where we interleave a potentially long-running request with a timer; we cancel the request if it takes too long. We can also run two identical requests to different services and just take the result of the one that returns first. The **`interleave`** operator can be composed with itself to yield the *n*-ary operator **`interleave`** operator.

Chapter 7

Conclusion

We conclude with a discussion of the achievements, some limitations, and possible future work.

Combining the pre-emptive concurrency of Section **??** with the promise library of Section **??** yields a direct and comfortable way to write asynchronous programs. It is our hope that these results will be reproducible in other effect-handling languages. Moreover, we have shown how effect handlers as a programming tool can be used to easily express complicated control flow.

TODO: Add more to end of paragraph

Limitations and Future Work The implementation of asynchronous effects as discussed does not track asynchronous effects, and is untyped. Ahman and Pretnar have shown that this is possible; it remains as future work to add types to our implementation.

Our system uses the fairly simple communication protocol where every message gets sent to every thread. Naturally, two threads might want to communicate secretly, without other threads eavesdropping. Finer control of this is desirable for larger applications.

Future Work

Bibliography

Danel Ahman and Matija Pretnar. Asynchronous effects. *arXiv preprint arXiv:2003.02110*, 2020.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *J. Funct. Program.*, 30:e9, 2020.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, page 13, 2015.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *International Symposium on Trends in Functional Programming*, pages 98–117. Springer, 2017.

Daniel Hillerström. Compilation of effect handlers and their applications in concurrency. *MSc (R) thesis, School of Informatics, The University of Edinburgh*, 2016.

Daan Leijen. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061*, 2014.

Daan Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, pages 16–29, 2017.

Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *arXiv preprint arXiv:1312.1399*, 2013.

Appendix A

Remaining Formalisms

$\boxed{\Phi; \Gamma\, [\Sigma]\!\vdash m \Rightarrow A}$

T-VAR
$$\frac{x : A \in \Gamma}{\Phi; \Gamma\, [\Sigma]\!\vdash x \Rightarrow A}$$

T-POLYVAR
$$\frac{\Phi \vdash \overline{R} \qquad f : \forall \overline{Z}.A \in \Gamma}{\Phi; \Gamma\, [\Sigma]\!\vdash f\ \overline{R} \Rightarrow A[\overline{R}/\overline{Z}]}$$

T-APP
$$\frac{\begin{array}{c}\Sigma' = \Sigma \qquad (\Sigma \vdash \Delta_i \dashv \Sigma'_i)_i \\ \Phi; \Gamma\, [\Sigma]\!\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to}\ [\Sigma']B\} \qquad (\Phi; \Gamma\, [\Sigma'_i]\!\vdash n_i : A_i)_i\end{array}}{\Phi; \Gamma\, [\Sigma]\!\vdash m\ \overline{n} \Rightarrow B}$$

T-ASCRIBE
$$\frac{\Phi; \Gamma\, [\Sigma]\!\vdash n : A}{\Phi; \Gamma\, [\Sigma]\!\vdash \uparrow\!(n : A) \Rightarrow A}$$

$\boxed{\Phi; \Gamma\, [\Sigma]\!\vdash n : A}$

T-SWITCH
$$\frac{\Phi; \Gamma\, [\Sigma]\!\vdash m \Rightarrow A \qquad A = B}{\Phi; \Gamma\, [\Sigma]\!\vdash \downarrow\! m : B}$$

T-DATA
$$\frac{k\ \overline{A} \in D\ \overline{R} \qquad (\Phi; \Gamma\, [\Sigma]\!\vdash n_j : A_j)_j}{\Phi; \Gamma\, [\Sigma]\!\vdash k\ \overline{n} : D\ \overline{R}}$$

T-COMMAND
$$\frac{\Phi \vdash \overline{R} \qquad c : \forall \overline{Z}.\overline{A \to}\ B \in \Sigma \qquad (\Phi; \Gamma\, [\Sigma]\!\vdash n_j : A_j[\overline{R}/\overline{Z}])_j}{\Phi; \Gamma\, [\Sigma]\!\vdash c\ \overline{R}\ \overline{n} : B[\overline{R}/\overline{Z}]}$$

T-THUNK
$$\frac{\Phi; \Gamma \vdash e : C}{\Phi; \Gamma\, [\Sigma]\!\vdash \{e\} : \{C\}}$$

T-LET
$$\frac{\begin{array}{c}P = \forall \overline{Z}.A \\ \Phi, \overline{Z}; \Gamma\, [\emptyset]\!\vdash n : A \qquad \Phi; \Gamma, f : P\, [\Sigma]\!\vdash n' : B\end{array}}{\Phi; \Gamma\, [\Sigma]\!\vdash \mathbf{let}\ f : P = n\ \mathbf{in}\ n' : B}$$

T-LETREC
$$\frac{\begin{array}{c}(P_i = \forall \overline{Z}_i.\{C_i\})_i \\ (\Phi, \overline{Z}_i; \Gamma, \overline{f : P} \vdash e_i : C)_i \qquad \Phi; \Gamma, \overline{f : P}\, [\Sigma]\!\vdash n : B\end{array}}{\Phi; \Gamma\, [\Sigma]\!\vdash \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n : B}$$

T-ADAPT
$$\frac{\Sigma \vdash \Theta \dashv \Sigma' \qquad \Phi; \Gamma\, [\Sigma']\!\vdash n : A}{\Phi; \Gamma\, [\Sigma]\!\vdash \langle\Theta\rangle\ n : A}$$

$\boxed{\Phi; \Gamma \vdash e : C}$

T-COMP
$$\frac{\begin{array}{c}(\Phi \vdash r_{i,j} : T_j \dashv\![\Sigma]\ \exists \Psi_{i,j}.\Gamma'_{i,j})_{i,j} \\ (\Phi, (\Psi_{i,j})_j; \Gamma, (\Gamma'_{i,j})_j\, [\Sigma]\!\vdash n_i : B)_i \qquad ((r_{i,j})_i \text{ covers } T_j)_j\end{array}}{\Phi; \Gamma \vdash ((r_{i,j})_j \mapsto n_i)_i : (T_j \to)_j\ [\Sigma]B}$$

Figure A.1: Term Typing Rules

$$\boxed{\Sigma \vdash \Delta \dashv \Sigma'}$$

$$\textsc{A-Adj}\quad \frac{\Sigma \vdash \Theta \dashv \Sigma' \qquad \Sigma' \vdash \Xi \dashv \Sigma''}{\Sigma \vdash \Theta \mid \Xi \dashv \Sigma''}$$

$$\boxed{\Sigma \vdash \Xi \dashv \Sigma'}$$

$$\textsc{A-Ext-Id}\quad \frac{}{\Sigma \vdash \iota \dashv \Sigma} \qquad \textsc{A-Ext-Snoc}\quad \frac{\Sigma \vdash \Xi \dashv \Sigma'}{\Sigma \vdash \Xi, I\,\overline{R} \dashv \Sigma', I\,\overline{R}}$$

$$\boxed{\Sigma \vdash \Theta \dashv \Sigma'}$$

$$\textsc{A-Adapt-Id}\quad \frac{}{\Sigma \vdash \iota \dashv \Sigma} \qquad \textsc{A-Adapt-Snoc}\quad \frac{\Sigma \vdash \Theta \dashv \Sigma' \qquad \Sigma' \vdash I(S \to S') \dashv \Sigma''}{\Sigma \vdash \Theta, I(S \to S') \dashv \Sigma''}$$

$$\boxed{\Sigma \vdash I(S \to S') \dashv \Sigma'}$$

$$\textsc{A-Adapt-Com}\quad \frac{\Sigma \vdash S : I \dashv \Sigma'; \Omega \qquad \Omega \vdash S' : I \dashv \Xi \qquad \Sigma' \vdash \Xi \dashv \Sigma''}{\Sigma \vdash I(S \to S') \dashv \Sigma''}$$

$$\boxed{\Sigma \vdash S : I \dashv \Sigma'; \Omega}$$

$$\textsc{I-Pat-Id}\quad \frac{}{\Sigma \vdash s : I \dashv \Sigma; s : \Sigma} \qquad \textsc{I-Pat-Bind}\quad \frac{\Sigma \vdash S : I \dashv \Sigma'; \Omega}{\Sigma, I\,\overline{R} \vdash S\,a : I \dashv \Sigma'; \Omega, a : I\,\overline{R}} \qquad \textsc{I-Pat-Skip}\quad \frac{\Sigma \vdash S\,a : I \dashv \Sigma'; \Omega \qquad I \neq I'}{\Sigma, I'\,\overline{R} \vdash S\,a : I \dashv \Sigma', I'\,\overline{R}; \Omega}$$

$$\boxed{\Omega \vdash S : I \dashv \Xi}$$

$$\textsc{I-Inst-Id}\quad \frac{s \in \mathsf{dom}(\Omega)}{\Omega \vdash s : I \dashv \iota} \qquad \textsc{I-Inst-Lkp}\quad \frac{a \in \mathsf{dom}(\Omega) \qquad \Omega \vdash S : I \dashv \Xi \qquad \Omega(a) = I\,\overline{R}}{\Omega \vdash S\,a : I \dashv \Xi, I\,\overline{R}}$$

Figure A.3: Action of an Adjustment on an Ability and Auxiliary Judgements

$$\mathcal{X} ::= A \mid C \mid T \mid G \mid Z \mid R \mid P \mid \sigma \mid \Sigma \mid \Xi \mid \Theta \mid \Delta \mid \Gamma \mid \exists\Psi.\Gamma \mid \Omega$$

$\boxed{\Phi \vdash \mathcal{X}}$

$$\frac{}{\Phi, X \vdash X}\ \textsc{WF-Val} \qquad \frac{}{\Phi, [E] \vdash E}\ \textsc{WF-Eff} \qquad \frac{\Phi, \overline{Z} \vdash A}{\Phi \vdash \forall \overline{Z}.A}\ \textsc{WF-Poly}$$

$$\frac{(\Phi \vdash R)_i}{\Phi \vdash D\,\overline{R}}\ \textsc{WF-Data} \qquad \frac{\Phi \vdash C}{\Phi \vdash \{C\}}\ \textsc{WF-Thunk} \qquad \frac{(\Phi \vdash T)_i \qquad \Phi \vdash G}{\Phi \vdash \overline{T \to} G}\ \textsc{WF-Comp} \qquad \frac{\Phi \vdash \Delta \qquad \Phi \vdash A}{\Phi \vdash \langle\Delta\rangle A}\ \textsc{WF-Arg}$$

$$\frac{\Phi \vdash \Sigma \qquad \Phi \vdash A}{\Phi \vdash [\Sigma]A}\ \textsc{WF-Ret} \qquad \frac{\Phi \vdash \Sigma}{\Phi \vdash [\Sigma]}\ \textsc{WF-Ability} \qquad \frac{}{\Phi \vdash \emptyset}\ \textsc{WF-Pure} \qquad \frac{}{\Phi \vdash \iota}\ \textsc{WF-Id} \qquad \frac{\Phi \vdash \Xi \qquad (\Phi \vdash R)_i}{\Phi \vdash \Xi, I\,\overline{R}}\ \textsc{WF-Ext}$$

$$\frac{\Phi \vdash \Theta}{\Phi \vdash \Theta, I\ (S \to S')}\ \textsc{WF-Adapt}$$

$$\frac{}{\Phi \vdash \cdot}\ \textsc{WF-Empty} \qquad \frac{\Phi \vdash \Gamma \qquad \Phi \vdash A}{\Phi \vdash \Gamma, x : A}\ \textsc{WF-Mono} \qquad \frac{\Phi \vdash \Gamma \qquad \Phi \vdash P}{\Phi \vdash \Gamma, f : P}\ \textsc{WF-Poly}$$

$$\frac{\Phi, \Psi \vdash \Gamma}{\Phi \vdash \exists\Psi.\Gamma}\ \textsc{WF-Existential} \qquad \frac{\Phi \vdash \Omega \qquad (\Phi \vdash R)_i}{\Phi \vdash \Omega, x : I\,\overline{R}}\ \textsc{WF-Interface}$$

Figure A.4: Well-Formedness Rules

$$\boxed{\Phi \vdash p : A \dashv \Gamma}$$

$$\textsc{P-Var}\quad \frac{}{\Phi \vdash x : A \dashv x : A}
\qquad\qquad
\textsc{P-Data}\quad \frac{k\,\overline{A} \in D\,\overline{R} \qquad (\Phi \vdash p_i : A_i \dashv \Gamma)_i}{\Phi \vdash k\,\overline{p} : D\,\overline{R} \dashv \overline{\Gamma}}$$

$$\boxed{\Phi \vdash r : T \dashv_{[\Sigma]} \exists \Psi.\Gamma}$$

$$\textsc{P-Value}\quad \frac{\Sigma \vdash \Delta \dashv \Sigma' \qquad \Phi \vdash p : A \dashv \Gamma}{\Phi \vdash p : \langle \Delta \rangle A \dashv_{[\Sigma]} \Gamma}
\qquad\qquad
\textsc{P-CatchAll}\quad \frac{\Sigma \vdash \Delta \dashv \Sigma'}{\Phi \vdash \langle x \rangle : \langle \Delta \rangle A \dashv_{[\Sigma]} x : \{[\Sigma']A\}}$$

$$\textsc{P-Command}\quad \frac{\Sigma \vdash \Delta \dashv \Sigma' \qquad \Delta = \Theta \mid \Xi \qquad c : \forall \overline{Z}.\overline{A \to} B \in \Xi \qquad (\Phi, \overline{Z} \vdash p_i : A_i \dashv \Gamma_i)_i}{\Phi \vdash \langle c\,\overline{p} \to z \rangle : \langle \Delta \rangle B' \dashv_{[\Sigma]} \exists \overline{Z}.\overline{\Gamma}, z : \{\langle \iota \mid \iota \rangle B \to [\Sigma']B'\}}$$

Figure A.5: Pattern Matching Typing Rules