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 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 effectful operation is invoked, the rest of the computation pauses whilst the effect handler performs the requisite computation and then resumes the original caller.

TODO: Do we need to speak about how the continuation of the caller gets 'offered up'?

For many effects, this blocking behaviour is not a problem; the handler usually returns quickly, and the user does not notice anything. Of course, 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.

TODO: Talk about extra examples...

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

TFP, aeff, etc

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 parts, 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

TODO: Fix above

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 also 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 the language, and show why it is so well-suited to our task. We assume some familiarity with typed functional programming, and skip over some more traditional aspects of the language — algebraic data types, pattern matching, etc — so we can spend more time with the novel, interesting parts of the language.

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*. Computation types are used to define an operator; they come in the general form `{<Effect> X -> ... -> [Effects] Y}`. This type expresses that the operator can handle some effect in the first argument and then performs some other effects as a result, returning a value of type `Y`.

Frank then specialises effect handling to traditional function application; a function

is the special case of an operator that handles no arguments. We see that a function type `{X -> Y -> Z}` is just a special case of the general operator type where no effects are handled or performed. Throughout this thesis, we call

Thunks then are the special case of an n-ary function that takes 0 arguments. A thunk, being a suspended computation, that will result in a value of type `Bool` has type `{Bool}`. This gives us fine-grained control over executing computations, and also translates to control over when to execute effects; this proves to be very useful later.

TODO: Example — maybe fire missiles one?

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`. Programs get access to an interface's command 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 Traditioanl functions in Frank are a specialisation of Frank's handlers; that is to say, functions are handlers that handle no effects. A handler for an interface pattern matches *on the operations* that are invoked, as well as on the *values* that the computation can return. Furthermore, the handler gets access to the *continuation* of the calling function as a first-class value. Consider the handler for `State`;

```
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
```

The type of `runState` expresses that the first argument is a computation that can perform `State S` effects and will eventually return a value of type `X`, whilst the second argument is a value of type `S`. The `State S` effect is then *removed* from the first argument.

What happens when we run `runState xplusplus! 0`? When a computation is invoked, it is performed until it results in either a *value* or a *command*. Thus, `runState` will be paused until `xplusplus!` reduces; `runState` is resumed when `xplusplus` is in one of these two forms.

`xplusplus` instantly invokes `get!`. At this point, control is given to the handler `runState`; both in the sense that `runState` is now being executed by the interpreter, and that `runState` has control over the *continuation* of `xplusplus`, which is a function of type `Int -> [State Int] Unit`. We see that `runState` chooses to resume this continuation with the value of the state at that time.

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. This can be characterised as *synchronous* effect handling.

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 asynchrony is exactly what we search for 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
3 interface Receive X = receive : X
4
5 pipe : {<Send X>Unit -> <Receive X>Y -> [Abort]Y}
6 pipe <send x -> s> <receive -> r> = pipe (s unit) (r x)
7 pipe <_> y = y
8 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. Hence we have a broken pipe, so the `abort` command is invoked. This can then be caught by another handler, which can implement a recovery strategy.

TODO: Is it worth changing the example to match request on the left earlier?

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 Cooperative Concurrency

Frank is a single-threaded language. It is fortunate, then, that effect handlers give us a malleable way to run multiple program-threads "simultaneously"

TODO: This is poorly written — fix

.

This is because the invocation of an operation not only offers up the operation's payload, but also the *continuation* of the calling computation. 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; in the case of `getState`, it is resumed with the value in state. But these continuations are first-class; they can resumed, sure, but also stored elsewhere or even thrown away. As such, by handling `Yield` operations, we easily pause and switch between several threads.

2.3.1 Simple Scheduling

We introduce some simple program threads 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 — of course, it is the way we *handle* yield that is more interesting.

```
-- Runs all of the LHS first, then the RHS.
scheduleA : {<Yield> Unit -> <Yield> Unit -> Unit}
```

```
scheduleA <yield -> m> <n> = scheduleA (m unit) n!
scheduleA <m> <yield -> n> = scheduleA m! (n unit)
scheduleA _ _ = unit

-- Lets two yields synchronise, then handles both
scheduleB : {<Yield> Unit -> <Yield> Unit -> Unit}
scheduleB <yield -> m> <yield -> n> = scheduleB (m unit) (n unit)
scheduleB <yield -> m> <n> = scheduleB (m unit) n!
scheduleB <m> <yield -> n> = scheduleB m! (n unit)
scheduleB _ _ = unit
```

TODO: Can maybe delete the 2nd and 3rd matches of scheduleB to make the point more clear?

We see two multihandlers above. Each take two `yield`ing threads and schedule them, letting one run at a time. `scheduleA` runs the first thread to completion, and only then runs the second one; the first time that the second thread `yield`s it is *blocked*, and can no longer execute. As such, the output of `scheduleA words! numbers!` is `one 1 two three 2 3 unit`.

`scheduleB` is fairer and more profound. We run `scheduleB words! number!` and receive `one 1 two 2 three 3 unit`; `scheduleB` is fair and will "match" the yields together. We step through slowly. First `words!` will print `one`, then it will `yield`. At this point — recalling that multihandlers pattern match left-to-right — the second thread, `numbers!`, is allowed to execute. In the meantime, `words!` is stuck as `<yield -> m>`; it cannot evaluate any further, it is *blocked*. Whilst `words` is blocked `numbers!` prints `1` and then `yield`s. Great; now the first case matches. Both threads are resumed and the process repeats itself.

TODO: The second paragraph here is a more compelling explanation; maybe we can just get rid of all of the scheduleA business and /just/ have the scheduleB stuff? scheduleA is quite obvious i think whilst B is more subtle and compelling.

TODO: It's not true that it matches L-R as much as runs all computations L - R until they are all a command / value - fix this

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
             | exit : 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! ";
          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 finishing executing the current thread, or eagerly run them, suspending the currently executing thread and running the forked process. The handler for the former, breadth-first style of scheduling, is

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

Notice the use of the `Queue` effect; we have to handle the computation `scheduleBF forker!` with a handler for `Queue` effects afterwards. 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.

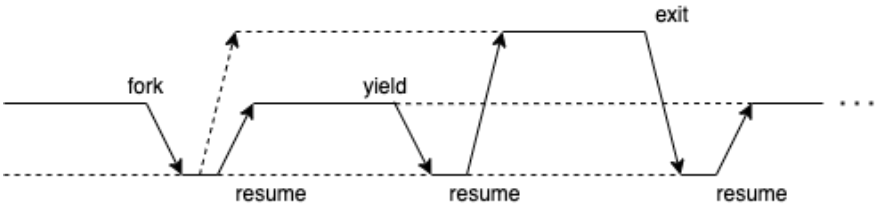

Figure 2.1: Breadth-First scheduling

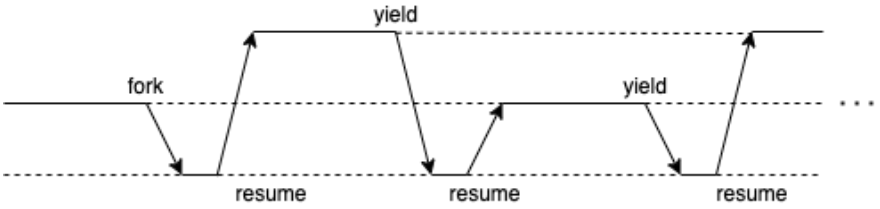

Figure 2.2: Depth-First scheduling

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 to get pre-emptive concurrency, we explain some of the key parts of the language in this section.

$$
\begin{array}{lll}
\text{(data types)} & D & \\
\text{(value type variables)} & X & \\
\text{(effect type variables)} & E| & \\
\text{(value types)} & A,B ::= & D\,\overline{R} \\
 & & \mid \ \{C\} \mid X \\
\text{(computation types)} & C ::= & \overline{T \to} G \\
\text{(argument types)} & T ::= & \langle\Delta\rangle A \\
\text{(return types)} & G ::= & [\Sigma]A \\
\text{(type binders)} & Z ::= & X \mid [E] \\
\text{(type arguments)} & R ::= & A \mid [\Sigma] \\
\text{(polytypes)} & P ::= & \forall\overline{Z}.A
\end{array}
$$

$$
\begin{array}{lll}
\text{(interfaces)} & I & \\
\text{(term variables)} & x,y,z,f & \\
\text{(instance variables)} & s,a,b,c & \\
\text{(seeds)} & \sigma ::= & \emptyset \mid E \\
\text{(abilities)} & \Sigma ::= & \sigma \mid \Xi \\
\text{(extensions)} & \Xi ::= & \iota \mid \Xi, I\,\overline{R} \\
\text{(adaptors)} & \Theta ::= & \iota \mid \Theta, I(S \to S') \\
\text{(adjustments)} & \Delta ::= & \Theta \mid \Xi \\
\text{(instance patterns)} & S ::= & s \mid S\ a \\
\text{(kind environments)} & \Phi,\Psi ::= & \cdot \mid \Phi, Z \\
\text{(type environments)} & \Gamma ::= & \cdot \mid \Gamma, x : A \mid \Gamma, f : P \\
\text{(instance environments)} & \Omega ::= & s : \Sigma \mid \Omega, a : I\,\overline{R}
\end{array}
$$

Figure 3.1: Types

Value types are either datatypes instantiated with type arguments $D\ \overline{R}$, thunked computations $\{C\}$, or value type variables X. Computation types are zero or more argument types T resulting in a return type G. A computation of type

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

Figure 3.2: Terms

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

has arugment types $\langle \Theta_i \mid \Xi_i \rangle A_i$ and return type $[\Sigma]B$; that is, a computation of type C handles effects in extensions Ξ_i in its i-hth argument. It then returns a value of type B and potentially performs effects in Σ.

TODO: Talk about adaptors at each index?

An ability Σ .... It may be closed $\emptyset$ or open E. An extension Ξ is a finite list of interfaces.

We deliberately leave out details on adaptors for the sake of brevity. We also skip over the typing rules, as they are standard. These can be seen in the appendix.

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 $\textbf{let}\ f : P = n\ \textbf{in}\ n'$, recursive let $\textbf{letrec}\ \overline{f : P = e}\ \textbf{in}\ n$ and adaptors $\langle \Theta \rangle\ n$. We can inject a use into a construction with $\downarrow$ and vice versa ($\uparrow$); in real Frank code these are not present.

Computations are produced by defining a sequence of pattern matching clauses. Each pattern matching clause takes a sequence $\overline{r}$ of computation patterns. These can

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

Figure 3.3: Runtime Syntax

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.

TODO: Talk about typing for Frozen commands — basically jus say that they retain the type when frozen.

Runtime Syntax The operational semantics uses the runtime syntax of Figure 3.3. The uses and constructions are supplemented with a special term $\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil$, of *frozen commands*. We discuss these further later.

We distinguish use and construction values, and then further separate construction values into uses and non-uses. We also declare a new class of *normal forms*, to be used in pattern binding.

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 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). We discuss these rules in more detail later. $r : T \leftarrow t \dashv [\Sigma]\ \theta$ states that the computation pattern r of type T and 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.

TODO: Tighten up description.

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 shows how commands, once used, 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 command c; i.e. it is a context of the form $u\ (\bar{t}, [\,], \overline{u'})$ 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.

TODO: Frozen commands - delimited continuations

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

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

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

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

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

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

R-FREEZE-FRAME-CONS

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

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

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

$$\textsc{B-Value}\ \frac{\Sigma \vdash \Delta \dashv \Sigma' \qquad p : A \leftarrow w \dashv \theta}{p : \langle \Delta \rangle A \leftarrow w \dashv [\Sigma]\ \theta}$$

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

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

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

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

$$\textsc{B-Var}\ \frac{}{x : A \leftarrow w \dashv [\uparrow(w : A)/x]} \qquad \textsc{B-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad (p_i : A_i \leftarrow w_i \dashv \theta_i)_i}{k\ \overline{p} : D\ \overline{R} \leftarrow k\ \overline{w} \dashv \overline{\theta}}$$

Figure 3.5: Pattern Binding

Chapter 4

Pre-emptive Concurrency

4.1 Motivation

One important part of our asynchronous effect handling system is the ability to interrupt arbitrary computations. If two threads are running concurrently and are communicating with one another, we have to stop running one to let messages come in from the other. This is achievable with explicitly yielding, as in Section 2.3, however we would prefer for this behaviour to be done automatically.

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`.

4.2 Interruption with Yields

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.

4.3 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 4.2. 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`.

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;

$$\text{B-CatchAll-Request-Loose}\quad \frac{\Sigma \vdash \Delta \dashv \Sigma'}{\langle x \rangle : \langle \Delta \rangle A \leftarrow \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil \dashv [\Sigma]\ [\uparrow(\{\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil\} : \{[\Sigma']A\})/x]}$$

Figure 4.1: Updated B-CatchAll-Request

```
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.

4.4 Freezing Arbitrary Terms

The approach of Section 4.3 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,

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

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

Figure 4.2: Runtime Syntax, Updated with Freezing of Uses

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

$$
\frac{}{m \rightsquigarrow_{\mathrm{u}} \lceil m \rceil}\ \text{R-FREEZE-USE}
\qquad
\frac{\mathcal{F} \text{ not handler}}{\mathcal{F}[\mathcal{E}[\lceil m \rceil]] \rightsquigarrow_{\mathrm{u}} \lceil \mathcal{F}[\mathcal{E}[m]] \rceil}\ \text{R-FREEZE-FRAME-USE}
$$

$$
\frac{\mathcal{F} \text{ not handler}}{\mathcal{F}[\mathcal{E}[\lceil m \rceil]] \rightsquigarrow_{\mathrm{c}} \lceil \mathcal{F}[\mathcal{E}[m]] \rceil}\ \text{R-FREEZE-FRAME-CONS}
$$

Figure 4.3: Updated Freezing

at which point the term is unfrozen and resumed. This process of freezing up to a handler is enforced by the predicate $\mathcal{F}$ not handler , which is true only when $\mathcal{F}$ is of the form $u\ (\overline{t}, [\,], \overline{n})$.

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 4.3. 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.

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

Figure 4.4: Catching Interrupts rule.

4.5 Yielding

Observe that the freezing approach of Section 4.4 ends up reimplementing a lot of the behaviour of the freezing of ordinary commands, without adding much new behaviour. 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

$n \rightsquigarrow_{\mathrm{u}} n'$

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

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 $c = \mathsf{false}$. In this case, it is defined as follows;

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

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 4.4, another being that we define fewer new rules and constructs.

Nondeterminism Observe that this system, and the system from Section 4.4, 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.

4.6 Counting

The semantics given by Section 4.5 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.

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

R-HANDLE-COUNT

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

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

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

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

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 **??**, 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 $\mathbb{F}_{\mathcal{C}}$ *implements* $\mathbb{F}_{\mathcal{ND}}$; that is to say the counting system gives a deterministic way to perform the nondeterministic system. We can formally state this follows.

Theorem 1 (Counting Implements Nondeterminism)

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

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

4.7 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.

4.8 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

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

Assume that

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

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 $\Phi;\Gamma\,[\Sigma'_{|\bar{t}|}]\vdash \mathcal{E}'[n] : A_{|\bar{t}|}$ and $\Sigma \vdash \Delta_{|\bar{t}|} \dashv \Sigma_{|\bar{t}|}$. It follows than that $\Phi;\Gamma\,[\Sigma'_{|\bar{t}|}]\vdash \mathcal{E}'[\mathsf{yield}; n] : A_{|\bar{t}|}$, 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 $\Phi;\Gamma\,[\Sigma]\vdash \mathcal{F}[n] : A$, and therefore $\Phi;\Gamma\,[\Sigma]\vdash n : A'$. By the assumption and subject reduction, $\Phi;\Gamma\,[\Sigma]\vdash n' : A'$. Then clearly $\Phi;\Gamma\,[\Sigma]\vdash \mathcal{F}[n'] : A$.

Theorem 3 (Type Soundness)

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

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

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 in the same way to an effect handler, but it does not get access to the continuation of the calling code.

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.

5.1 In Frank

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.

The **`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`**

[1] Interrupt handlers which put conditions on the incoming interrupts are called *guarded* interrupt handlers — we come back to these later.

`| result ResultType`. The handler for `Promise` will then send the signals to each other thread, possibly executing the interrupt handler if needs be.

`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

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

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** 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.

5.2 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.

5.3 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

6.1 Pre-emptive Concurrency

An essential feature of our asynchronous effects system is that it supports pre-emptive concurrency; that is, the suspension and resumption of threads non-cooperatively. Naturally, this relies on the insertion of yields as discussed in Chapter **??**.

We supplement the signals supported in our program with two more;

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

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[Console], Console, 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[Console], Console,
   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 id of the thread. The body is then fairly simple; it installs a

promise waiting for `go` and immediately starts blocking. The rest of the computation can not proceed until the corresponding go message is received. Once the go promise is fulfilled, the non-blocking stop promise is reinstalled.

```
goPromise : {Int -> Sig -> Maybe {[Promise[Console], Console, 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 make a function pre-emptible by just installing a stop-waiting promise in front of the function code;

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

thread1 : {[Promise[Console], Console, RefState, Yield] Unit}
thread1! = promise (prom {s -> stopPromise 0 s}); counter 0
```

Observe that all we have to do is precompose with the promise installer; the rest of the code goes on unaware that it is being pre-empted. Threads can also then communicate etc on top of this.

TODO: Is this example even interesting now that we've already got it baked into the language?

6.2 Async-Await

Here we show how our asynchronous effects system can express the familiar async-await abstraction.

Consider that we want to asynchronously run web requests using the built-in `getRequest` operation. These return a value of type `String`, being hte result of the request. First we add two more signals to our set of available signals;

```
data Sig = ... | call {String} Int | result String Int
```

TODO: Observe that this is a higher-order effect - something aeff lacks!

The signal `call` is used to start an asynchronous operation; the thunked argument is the computation we want to run. `result` is the signal used by the running thread to

indicate that it has completed the computation and is returning the **`String`** result. The **`Int`** arguments are for call IDs, so that the wrong results are not re-read.

Unlike other implementations, the Frank realisation of async-await does not dynamically create new threads to run asynchronous tasks. Instead, we have a dedicated thread that only performs these asynced processes. This may seem inefficient, however see that when not executing a process the thread will be instantly skipped in the scheduler, so we have no overhead costs.

We how show the **`async`** function that a caller would use to issue a new asynchronous task;

```
resultWaiter : {Int -> [Promise[Web,
    Console]] Pid String [WebThreads]
resultWaiter callNo =
  promise (prom { (result res callNo') -> if (callNo == callNo') {
      just {res} } {nothing}
                  | _ -> nothing})

async : {[Console, Web] String} -> Ref Int ->
    [WebThreads] Pid String [WebThreads]
async proc callCounter =
    let callNo = read callCounter in
    let waiter =
      <Console, RefState, Web, Yield>(resultWaiter callNo) in
    signal (call proc callNo);
    write callCounter (callNo + 1);
    waiter
```

So **`async`** takes the process to be run and a reference to the callcounter. It then installs another promise, **`resultWaiter`**, which waits for the corresponding **`result`** signal to be received. **`resultWaiter`** is an example of a *guarded* interrupt handler; it only fires if a certain condition regarding to the signal's payload holds (i.e. that **`callNo == callNo'`**). After installing **`resultWaiter`**, **`async`** sends a **`call`** signal with the process and callNo as argument, increments the call counter and returns the result-waiting promise.

```
onRun : {{[Console, Web] String} -> Int -> [WebThreads] Unit}
onRun proc callId =
    let res = <Promise, RefState, Yield> proc! in
    signal (result res callId);
    <Console, RefState, Web, Yield> runner!;
    unit
```

```
runner : {[Promise[Web, Console]] Pid Unit [WebThreads]}
runner! =
    promise (prom {(call proc callId) -> just {onRun proc callId}
                  | _ -> nothing})
```

`runner` is the process that runs on the worker thread. This simply installs a promise that responds to `call` signals. On receipt of a `call` it runs the delivered process synchronously; once it is finished it sends a `result` signal and then finally reinvokes the `runner`. Note that `proc` can still have `yield` calls inserted into it, so that this doesn't cause the whole program to block.

TODO: Example of how it gets used.

6.3 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).

6.4 Cancelling Tasks

TODO: Link back to the async/await w/ dynamic

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 these tasks run on their own thread. As such, we can just cancel the entire thread at the top-level.

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 the rest of the computation and reinstalls the promise handler. We need to then wrap this in a handler for Cancel effects again; this is because user-level promises could perform Cancel effects. Thus, we convert them to

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

TODO: Talk about informing the scheduler that the thread has been cancelled as well.

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 **?**.

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

```
interleave : {{[InTask] String} -> {[InTask] String}
          -> Ref Int -> [WebEffs] Pid String [WebEffs]}
interleave procA procB callCounter =
    -- Issue the two calls with signals.
    let callNoA = read callCounter in
    write callCounter (callNoA + 1);
    let callNoB = read callCounter in
    write callCounter (callNoB + 1);

    -- Install a waiter to wait for the first result
    let ileaveWaiter = (leaveResWaiter callNoA callNoB) in

    -- Signal to start the other two
    signal (call procA callNoA);
    signal (call procB callNoB);

    -- Just return waiter.
    ileaveWaiter
```

This will set two threads running on independent threads. It then installs an interrupt handler for result messages; whichever result returns first, we cancel the other task and just return the result of the original one.

This lets us write timeouts for functions; we can cancel a task if it takes too long to return (e.g. a request to a web server that is inactive), or run two remote identical requests to different services and just take the result of whichever one returns first. This also generalises to the *n*-ary case in the natural way.

Chapter 7

Conclusion

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, 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.

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

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
$$\frac{\Sigma' = \Sigma \qquad (\Sigma \vdash \Delta_i \dashv \Sigma'_i)_i \qquad \Phi;\Gamma\,[\Sigma]\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to}\ [\Sigma']B\} \qquad (\Phi;\Gamma\,[\Sigma'_i]\vdash n_i : A_i)_i}{\Phi;\Gamma\,[\Sigma]\vdash m\ \overline{n} \Rightarrow B}$$

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
$$\frac{P = \forall \overline{Z}.A \qquad \Phi,\overline{Z};\Gamma\,[\emptyset]\vdash n : A \qquad \Phi;\Gamma, f : P\,[\Sigma]\vdash n' : B}{\Phi;\Gamma\,[\Sigma]\vdash \textbf{let}\ f : P = n\ \textbf{in}\ n' : B}$$

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

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
$$\frac{(\Phi \vdash r_{i,j} : T_j \dashv [\Sigma]\ \exists \Psi_{i,j}.\Gamma'_{i,j})_{i,j} \qquad (\Phi,(\Psi_{i,j})_j;\Gamma,(\Gamma'_{i,j})_j\,[\Sigma]\vdash n_i : B)_i \qquad ((r_{i,j})_i \text{ covers } T_j)_j}{\Phi;\Gamma \vdash ((r_{i,j})_j \mapsto n_i)_i : (T_j \to)_j\ [\Sigma]B}$$

Figure A.1: Term Typing Rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure A.4: Well-Formedness Rules

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

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

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

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

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

Figure A.5: Pattern Matching Typing Rules