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

1 Introduction **1**

1.1 Related Work 1

1.2 Contributions 1

2 Programming in Frank **2**

2.1 Types, Values and Operators 2

2.2 Effects and Effect Handling 3

2.3 Cooperative Concurrency 7

2.3.1 Simple Scheduling 7

2.3.2 Forking New Processes 8

3 Formalisation of Frank **11**

4 Pre-emption **17**

4.1 Motivation 17

4.2 Interruption with Yields 17

4.3 Relaxing Catches 18

4.4 Interrupting Arbitrary Terms 20

4.5 Interrupting In Practice 22

4.6 Soundness 23

5 Implementation **24**

5.1 In Frank 24

5.2 Handling Promises 26

5.3 Multithreading 29

6 Examples **30**

6.1 Pre-emptive Concurrency 30

6.2 Async-Await 31
6.3 Futures 33

7 Conclusion **35**

Bibliography **36**

A Remaining Formalisms **37**

Chapter 1

Introduction

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.

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.

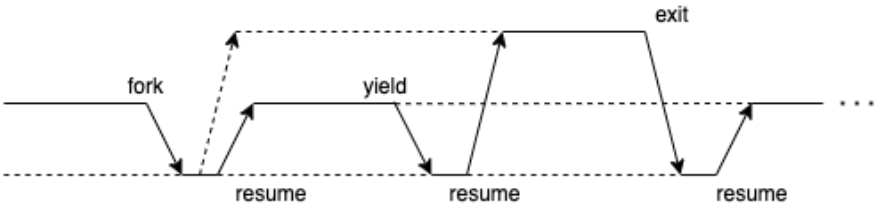

Figure 2.1: Breadth-First scheduling

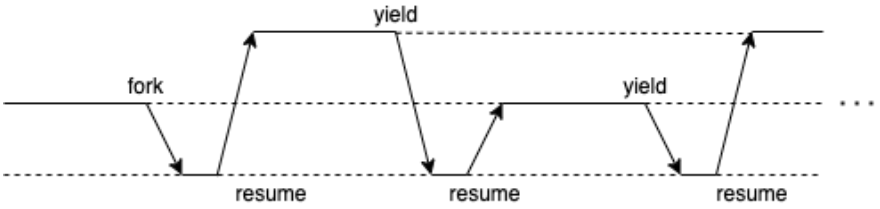

Figure 2.2: Depth-First scheduling

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.

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

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

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

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 (**?**); 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 either be a request pattern $\langle c\ \overline{p} \to z \rangle$, a catch-all pattern $\langle x \rangle$, or a standard value pattern

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

Figure 3.3: Runtime Syntax

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-emption

4.1 Motivation

One important part of our asynchronous effect handling system is the ability to interrupt arbitrary computations. This is essential for pre-emptive concurrency, which relies on being able to suspend a computation *non-cooperatively*; the computation gets suspended without being aware of its suspension.

TODO: Rewrite this - clumsy

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!
```

B-CATCHALL-REQUEST

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

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

```
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.

So far so good; this works as we hoped. However, observe that we had to change the code of the `runner` to `yield` every time it prints. This is not in the spirit of pre-emptive concurrency; we are still operating co-operatively. Threads should be unaware of the fact they are even being pre-empted. Furthermore, see that the `yield` operation adds no more information; it is just used as a placeholder operation; any operation would work. As such, we keep searching for a better solution.

TODO: Penultimate sentence could maybe go , a bit clumsy / weird

4.3 Relaxing Catches

The key to this lies in the catchall pattern, $\langle x \rangle$, and the pattern binding rules of Figure 3.5; specifically B-CATCHALL-REQUEST. We quickly go into detail on this rule

B-CATCHALL-REQUEST

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

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 4.2: Updated B-CATCHALL-REQUEST

now. $\langle x \rangle : \langle \Delta \rangle A$ states that $\langle x \rangle$ is a term with value type A and *adjustment* $\Delta = \Theta \mid \Xi$, made up of an adaptor Θ and an extension Ξ. This extension is made up of a list of interface instantiations $I\ \overline{R}$.

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 an element of this 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. The resulting rule B-CATCHALL-REQUEST-LOOSE can be seen in Figure 4.2. 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 unit})
suspend <_>          <go -> c>   (just res) =
    suspend res! (c unit) nothing
suspend unit         <_>         _ = unit
```

TODO: Check that the above still works...

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

Figure 4.3: Runtime Syntax, Updated with Freezing of Uses

Now when we run **`suspend runner! controller! nothing`**, the **`suspend`** handler is able to

TODO: Talk about how this still maintains the "no-snooping" policy; we can't inspect or access the effects that are invoked, but we know they happen.

Even with this extension,

4.4 Interrupting 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 **??**, we see an updated version of the runtime syntax; this allows for the suspension of arbitrary *uses*, essentially just function applications.

TODO: verify that uses are "just" apps and constructions

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.

With this in mind, we now give the updated rule for the catchall pattern matching on frozen terms. This can be seen in Figure 4.5. It expresses that an arbitrary frozen

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

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

R-FREEZE-USE

$$\frac{}{m \rightsquigarrow_{\mathrm{u}} \lceil m \rceil}$$

R-FREEZE-FRAME-USE

$$\frac{}{\mathcal{F}[\mathcal{E}[\lceil m \rceil]] \rightsquigarrow_{\mathrm{u}} \lceil \mathcal{F}[\mathcal{E}[m]] \rceil}$$

R-FREEZE-FRAME-CONS

$$\frac{}{\mathcal{F}[\mathcal{E}[\lceil m \rceil]] \rightsquigarrow_{\mathrm{c}} \lceil \mathcal{F}[\mathcal{E}[m]] \rceil}$$

Figure 4.4: Updated Freezing

TODO: Maybe need to add that the eval ctx is NOT a handler?

B-CATCHALL-INTERRUPT

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

Figure 4.5: Catching Interrupts rule.

use 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.

Figure 4.6 shows how uses m become interrupted. This rule supplements the operational semantics of Figure **??**. It states that at any point, a use term can reduce to the same term but suspended. At this point, the term cannot reduce any further; observe that $\lceil \; \rceil$ is not an evaluation context. The term then blocks until unfrozen. Note how similar this is to regular command invocations, which block until their continuation is invoked.

TODO: Check that just uses is enough

TODO: Talk about how this achieves our goal.

The addition of this rule introduces non-determinism into the language; at any point, a use can either step as normal (e.g. through the R-HANDLE rule), or it can be interrupted. An interrupted term $!(m)$ can no longer reduce; it is blocked until it is resumed by the R-HANDLE rule.

TODO: Talk about non-determinism as a result of this

TODO: Maybe move non-determinism to the next section?

$$\text{R-INTERRUPT}\quad \frac{}{m \rightsquigarrow_{\mathrm{u}} !(m)}$$

Figure 4.6: Use interruption rule

4.5 Interrupting In Practice

Due to the non-determinism introduced by R-INTERRUPT, this system is difficult to implement; how do we choose whether to apply a handler to its arguments or to just interrupt it?

In our implementation of this system, we instead maintain a counter which is incremented every time a handler is evaluated. When the counter reaches a certain threshold value t, we interrupt the current term m, applying the R-Interrupt rule. This converts the non-deterministic system to a deterministic one; we never have any question of *what* to do.

Observe that the process of interrupting a computation is a familiar one; we stop computing and offer up the continuation to the programmer. Where is a similar control flow already around? That's right — invocation of an effect.

TODO: Rewrite above paragraph to be less camp

As such, we can just use a normal effect to perform the interruption. Sticking with previous themes, we choose to invoke a `yield` effect when interrupting. This lets the programmer choose to handle the effect as they wish. Note that now interrupts are not restricted to the catch-all pattern $\langle m \rangle$ but are normal computation patterns.

These `yield` effects are only inserted in a computation when the `Yield` interface is present in the ability of this computation. This is important, as it lets the programmer get fine-grained control over which computations can be interrupted and which cannot. Consider the example in Section 4.1; we want the `runner` computation to be controller by the `controller`. As such, we want the `runner` to be interruptible, whilst the `controller` is not. We can reflect this by adding the `Yield` interface to the ability of the former computation.

Thus our example from before becomes;

TODO: updated example

4.6 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. [2020] with extra rules; as such we omit most of the details.

Theorem 1 (Subject Reduction)

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

By induction on the transitions $\rightsquigarrow_{\mathrm{u}}, \rightsquigarrow_{\mathrm{c}}$.

TODO: Finish this; addition of new yielding rule.

Theorem 2 (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 $\cdot;\cdot\,[\Sigma]\vdash n : A$.

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]). Æff is 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*. 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 traditional 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 does block computation if an answer is not already received.

Our system untyped in that there is no tracking of which asynchronous effects are issued in which functions, however it is typed in that the traditional Frank effects promises can perform are tracked.

Æff has an effect tracking system for asynchronous effects; our system does not. Ours does however track the effects that can be performed by interrupt handlers.

5.1 In Frank

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

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

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

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

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

We have a lot to unpack here, so we start slowly. 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]. The `promise` operation returns a `Pid R` (promise id), which is used to check if the installed promise has completed or not. The `R` type parameter is determined by the return type of the interrupt handler. This is later used in `await`.

`signal` is a more simple operation. The `Sig` data type is the type of signals that the thread can invoke. These can have extra bits of information — also called *payloads* — such as the parameters for a remote request. 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. If the promise has been fulfilled then the promise status stored in `Pid` will be `done res`; hence we just return this value. If it is incomplete — i.e. the promise status is `empty` — we add the resumption to `resume`.

Effect Typing We can track and control the effects that promises can perform using Frank's effect type system. For instance, a Frank program of type `[Promise [Console ]] X` can install promises that print to the console, a program of type `[Promise[ Console, Web]] X` can install promises that perform web requests and print to the console, etc.

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

Note however that the effect typing is slightly complicated in the definition of the interface; a type of `Promise [Console]` means that the installed promise can really perform the effects `[Promise[Console], Console, RefState, Yield]`. This is expressed by the `[E | Promise[E|], RefState, Yield]` part of the `promise R` definition. A recursive type is needed as the promises can themselves invoke other promises.

TODO: Flesh this out, rewrite it

Threads In order to run threads in parallel, we need to maintain a collection of thread states; when we stop executing a thread we store its continuation so far and start executing a new one, in the same style as Section 2.3.

We also need to store in this collection the promises that each thread has installed. These are stored as a stack so as to maintain the order of installation. Installing a promise is as straightforward as pushing it onto the corresponding thread's promise stack.

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

This collection is realised in Frank as the `Threads` datatype. It is essentially just an integer map, where keys are the thread IDs and the values are the trio of the thunked computation thus far, the stack of installed promises at this thread, and then the rest of the map.

The type of things called in the stack is not what we would expect. Recall that the type of Promises is `Sig -> Maybe R`, yet the promises stored here are of type `Sig -> Unit -> Maybe Unit`. We address this in Section 5.2, where it will become clearer.

5.2 Handling Promises

We now introduce the handler for `Promise` effects. The type of this comes as no surprise;

```
hdl : {Int -> Ref (Threads [E|])
```

```
    -> <Promise[E|]> Unit
    -> [E| RefState, Yield] Unit}
```

The first argument is the id of the thread being handled. The second one is a reference to the globally stored threads. 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).

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

Above we see the handler for `promise`. This is the process of installing a promise, onto the currently executing thread's stack. 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 `to_write`. This takes a callback of type `S -> R` and converts it to type `S -> [RefState]Unit`; it modifies it to always return unit but write its value to the `Pid` cell. Having all promises as the same type makes storing them in one data structure possible. In lines 4-6 we convert our promise to one that takes into account the computation after we've run the promise as well. This is essential to get blocking via `await` to work properly when running a promise, which is important for pre-emptive concurrency and other applications.

TODO: Go into more depth about this

.

Lines 7 to 10 are then mainly housekeeping. We write this modified promise into the stack, update the execution state thus far and then continue handling the thread.

TODO: Is all of the business with writing the rest of the callback needed?

TODO: It's not - can remove it.

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

```
    write thrs newThrs;
    hdl thId thrs (thr unit)
```

When a signal comes in, we need to try and execute *all* of the installed promises, for every thread. To do this we use the **`runThreads`** function, which calls the below function for all threads;

```
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 susp2) -> runThread sig (trio susp2 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. Recall that the type of the callbacks is **`Sig -> { Unit} -> {R}`**; we have to also supply the callbacks with the thunked computation so far. Again, this is important to correctly implement blocking. If the callback returns **`nothing`** we do not evaluate it and we reinstall it afterwards, by putting it onto the "skipped" stack.

Once promise execution is finished we update the state of **`thrs`** and resume handling, restarting the continuation **`thr`** with **`unit`** immediately.

```
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. If the promise has not yet completed, we then write the resumption (which is of type **`{R -> Unit}`**) to the cell. The function **`to_write`** used when installing promises changes the original promise to resume the continuation stored in **`Pid`**, if there is one present.

TODO: Should probably show to$_{w}rite$.

5.3 Multithreading

We now show how we can easily introduce multithreading to our system. Observe that we are yet to have a handler for any `yield` commands. We can handle them, yielding co-operative concurrency in the same style as Section 2.3, like so;

```
1 scheduleT : {<Yield> Unit -> Int -> Ref (Threads [E|]) -> [E|
    RefState]Unit}
2 scheduleT <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     scheduleT newThk! next thrs
8
9 scheduleT 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 the threads structure. Line 5 then writes the current thread's continuation to the structure. Line 6 writes the updated version of the threads and then 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 e.g. blocked and waiting for a promise to fulfill, as we will straightaway stop processing it and start the next one.

TODO: Talk about how we're still tied into the threads structure; disadvantage

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).

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.

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