Asynchronous Effect Handling

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. These features are often built into the language, and are opaque to the user.

In this thesis we show how a library for asynchronous programming can be very easily implemented 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.

Our system is expressive enough to define common asynchronous programming constructs, such as async-await and futures, within the language itself.

Acknowledgements

thanks!

Table of Contents

Chapter 1

Introduction

Effects, such as state and nondeterminism, are pervasive when programming; for a program to do anything beyond compute a pure mathematical function, it must interact with the outside world, be this to read from a file, make some random choice, or run concurrently with another program. Algebraic effects and their handlers (Plotkin and Pretnar [2013]) are a novel way to encapsulate, reason about and specify computational effects in programming languages. For instance, a program that reads from and writes to some local state can utilise the State effect, which supports two *operations*; get and put. A handler for the State effect gives a meaning to these abstract operations. Programming with algebraic effects and handlers is increasingly popular; they have seen adoption in the form of libraries for existing languages (Kammar et al. [2013], Kiselyov et al. [2013], Brady [2013]) as well as in novel languages designed with effect handling at their core (Bauer and Pretnar [2015], Leijen [2017b], Convent et al. [2020]).

Traditional effect handling is *synchronous*; when an operation is invoked, the rest of the computation pauses whilst the effect handler performs the requisite computation and then resumes the original caller. For many effects, this blocking behaviour is not a problem; the handler usually returns quickly, and the user notices no delay. However, not every possible computational effect behaves like this. Consider an effect involving a query to a remote database. We might not want to block the rest of the computation whilst we perform this, as the query might take a long time; this case is even stronger if we do not immediately want the data. To support this kind of behaviour, we need to be able to invoke and handle effects in an asynchronous, non-blocking manner.

In this project we investigate the implementation and applications of asynchronous effect handling. Our lens for this is the language Frank (Convent et al. [2020]), a func-

tional 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 by making a simple change to the semantics of Frank, in order to yield pre-emptible threads, we can recreate the asynchronous effect handling behaviour of Æff whilst still enjoying the benefits of traditional effect handlers.

TODO: Last paragraph could be fixed

Frank is well-suited to implement an asynchronous effects library. The fine-grained control over suspended computations makes it easy to treat code as data, and

TODO: Something else after the end

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.

TODO: Rewrite. Want to say that the change to the semantics is simple enough and the implementation simple enough that there should be no problems recreating our work in another language.

Effect handlers have already proven to make complicated control flow easy to implement (**refs**), and our work further cements this viewpoint.

Our contributions are as follows;

- We present a library for programming with asynchronous effects, built in Frank. We show how a complex system can be expressed concisely and elegantly when programming in a language with effect handlers.
- 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.
- We also deliver a set of examples of the uses of asynchronous effects, and show how they have benefits to other models.

1.1 Related Work

Asynchronous programming with effect handlers is a fairly nascent field. Koka (Leijen [2014]) is a programming language with built-in effect handlers and a Javascript back-

end. Leijen later shows us how Koka can naturally support asynchronous programming (Leijen [2017a]). The asynchronous behaviour relies on offloading asynchronous tasks with a setTimeout function supplied by the NodeJS backend.

Multicore OCaml (Dolan et al. [2014]) also supports asynchronous programming through effect handling (Dolan et al. [2017]). They handle effects and signals, which can be received asynchronously, and show how to efficiently and safely write concurrenct systems programs. However, in a similar way to Koka, the asynchrony relies on the operating system supplying operations, such as setSignal and timer signals.

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

Ahman and Pretnar then show how the simple building blocks of interrupt handlers can be used to build common constructs for asynchronous programming, such as cancellable remote function calls and a pre-emptive scheduler.

TODO: Section on the expressivity?

1.2 Structure

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

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

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

In Chapter 5 we describe the implementation of our asynchronous effect handling

library in Frank. In Chapter 6 we give examples of the new programs that become easily expressible when combined with the changes made in Chapter 4.

In Chapter 7 we conclude.

Chapter 2

Programming in Frank

Frank is a typed functional programming language, designed around the definition, control and handling of algebraic effects. As such, Frank has an effect type system used to track which effects a computation may use.

TODO: Extend this a bit - talk about ambient ability, computation vs. value, etc

In this chapter we introduce Frank, and show why it is so well-suited to our task. We assume some familiarity with typed functional programming, and skip over some common features of Frank — algebraic data types, pattern matching, etc. — so we can spend more time with the novel, interesting parts. We also skip some novel features of Frank, such as adaptors (Convent [2017]), as they are not essential for understanding the work of this project.

2.1 Types, Values and Operators

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

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

TODO: Go more into depth about 'abilities'?

Thunks are the special case of an n-ary function that takes 0 arguments. We can evaluate them — performing the suspended computation — with the 0-ary sequence of

arguments, denoted `!`. The opposite action — suspending a computation — is done by surrounding the computation in braces, such that for a suspended computation `comp` , `{comp!}` is the identity. This gives us fine-grained control over when we want to evaluate computations.

Consider the operator `badIf` below;

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

Frank is a *left-to-right*, *call-by-value* language; all arguments to operators are evaluated from left-to-right until they become a value. As such, in the case of `badIf`, both of the branches will be evaluated before the result of one of them is returned. We can recover the correct semantics for `if` by giving the branches as thunks;

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

Here a single thunk is evaluated depending on the value of the condition. Frank's distinction between computation and value make controlling evaluation simple and pleasing.

2.2 Effects and Effect Handling

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

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

interface RandInt = random : Int
```

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

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

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

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

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

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

Computation Patterns The second and third lines specify how we handle `get` and `put` commands. Observe that we use a new type of pattern, called a *computation pattern*; these are made up of a command and some arguments (which are also values), plus the continuation of the calling code. The types of arguments and the continuation are determined by the interface declaration and the type of the handler; for instance, in `<get -> k>` the type of `k` is `{S -> [State S] X}`. The continuation can then perform more `State S` effects. This differs to some other implementations of effect handling languages (Kammar et al. [2013]) where the handlers can be *deep*, meaning the continuation has been re-handled by the same handler automatically. Frank's *shallow* handlers mean we have to explicitly re-handle the continuation, but have the benefit of giving more control over how we would like to do so.

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

```
xplusrand : {[State Int, RandomInt] Unit}
xplusrand! = put (get! + 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` to generate pseudorandom numbers;

```
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 !)`.

Observe that the interaction between `xplusrand` and the handlers becomes like a conversation; the caller asks the handler for a result and waits, blocking, until the handler responds. We can characterise this as *synchronous* effect handling. But what if we want to make a request for information — such as the pseudorandom number — and do something else, then pick it up later? We cannot just invoke `random` as this would block whilst the number is generated, which could possibly take a long time. This *asynchronous* behaviour is exactly what we look for in this project.

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
```

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.

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

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

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

2.3 Case Study: Cooperative Concurrency

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

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

2.3.1 Simple Scheduling

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

```
interface Yield = yield : Unit

words : {[Console, Yield] Unit}
```

```
words! = print "one "; yield!; print "two "; yield!; print "three ";
    yield!

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

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

We can write a *multihandler* to schedule these two programs. A multihandler is simply an operator that handles multiple effects from different sources simultaneously.

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

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

If we omit line 2 we get quite a different result; the console output would be `one 1 two three 2 3`. This is because both threads are first evaluated until they are either a command or a value; this prints out `one 1`. Here we see the first use of the catch-all pattern `<n>`, which matches either a command or a value. At this point we resume the first thread, but the second thread remains blocked as the `yield` invocation has not been handled. We evaluate the first thread until it is `unit`, at which point we do the same to the second thread.

2.3.2 Forking New Processes

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

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

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

```
forker : {[Console, Co [Console]] Unit}
forker! = print "Starting! ";
          fork {print "one "; yield!; print "two "};
          fork {print "1 "; yield!; print "2 "};
          exit!
```

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

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

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

Chapter 3

Formalisation of Frank

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

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

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

Figure 3.1: Types

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

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

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

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

Figure 3.2: Terms

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

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

We omit details on adaptors as they are present in previous work (Convent et al. [2020]). The same goes for the typing rules, which we do not change.

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

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

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

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

Figure 3.3: Runtime Syntax

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

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

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

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

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

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

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

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

$$\text{R-ASCRIBE-USE}\quad \frac{}{\uparrow(\downarrow u : A) \rightsquigarrow_{\mathrm{u}} u} \qquad \text{R-ASCRIBE-CONS}\quad \frac{}{\downarrow\uparrow(w : A) \rightsquigarrow_{\mathrm{c}} w} \qquad \text{R-LET}\quad \frac{}{\mathbf{let}\ f : P = w\ \mathbf{in}\ n \rightsquigarrow_{\mathrm{c}} n[\uparrow(w : P)/f]}$$

$$\text{R-LETREC}\quad \frac{}{\mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n' \rightsquigarrow_{\mathrm{c}} n'[\uparrow(\{\overline{\overline{r} \to \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n}\} : P)/f]} \qquad \text{R-ADAPT}\quad \frac{}{\langle \Theta \rangle\ w \rightsquigarrow_{\mathrm{c}} w}$$

$$\text{R-FREEZE-COMM}\quad \frac{}{c\ \overline{R}\ \overline{w} \rightsquigarrow_{\mathrm{c}} \lceil c\ \overline{R}\ \overline{w} \rceil}$$

$$\text{R-FREEZE-FRAME-USE}\quad \frac{\neg(\mathcal{F}\,[\mathcal{E}]\ \mathsf{handles}\ c)}{\mathcal{F}\,[\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil] \rightsquigarrow_{\mathrm{u}} \lceil \mathcal{F}\,[\mathcal{E}[c\ \overline{R}\ \overline{w}]] \rceil} \qquad \text{R-FREEZE-FRAME-CONS}\quad \frac{\neg(\mathcal{F}\,[\mathcal{E}]\ \mathsf{handles}\ c)}{\mathcal{F}\,[\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil] \rightsquigarrow_{\mathrm{c}} \lceil \mathcal{F}\,[\mathcal{E}[c\ \overline{R}\ \overline{w}]] \rceil}$$

$$\text{R-LIFT-UU}\quad \frac{m \rightsquigarrow_{\mathrm{u}} m'}{\mathcal{E}[m] \longrightarrow_{\mathrm{u}} \mathcal{E}[m']} \qquad \text{R-LIFT-UC}\quad \frac{m \rightsquigarrow_{\mathrm{u}} m'}{\mathcal{E}[m] \longrightarrow_{\mathrm{c}} \mathcal{E}[m']} \qquad \text{R-LIFT-CU}\quad \frac{n \rightsquigarrow_{\mathrm{c}} n'}{\mathcal{E}[n] \longrightarrow_{\mathrm{u}} \mathcal{E}[n']} \qquad \text{R-LIFT-CC}\quad \frac{n \rightsquigarrow_{\mathrm{c}} n'}{\mathcal{E}[n] \longrightarrow_{\mathrm{c}} \mathcal{E}[n']}$$

Figure 3.4: Operational Semantics

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

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

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

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

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

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

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

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

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

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

Figure 3.5: Pattern Binding

Chapter 4

Pre-emptive Concurrency

4.1 Motivation

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

Consider the two programs below;

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

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

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

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

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

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

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

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

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

4.2 Relaxing Catches

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

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

B-CATCHALL-REQUEST-LOOSE

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

Figure 4.1: Updated B-CATCHALL-REQUEST

In the interests of pre-emption, we propose to remove this constraint from B-CATCHALL-REQUEST, replacing the rule with B-CATCHALL-REQUEST-LOOSE as

seen in Figure 4.1. This lets us update the previous **suspend** code to the following, which yields the same results as last time;

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

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

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

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

4.3 Freezing Arbitrary Terms

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

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

Note that frozen terms here behave in a similar way to frozen commands, by freezing the rest of the term around it as well. This continues up until a handler is reached, at which point the term is unfrozen and resumed. This process of freezing up to a handler is enforced by the predicate $\mathcal{F}$ 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

[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

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

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

Figure 4.3: Updated Freezing

term can be matched against the computation pattern $\langle x \rangle$. The suspended, unfrozen computation $\{m\}$ is then bound to x, in a similar way to other B-CATCHALL rules. Observe that this maintains no-snooping; we don't know that the frozen computation performed an effect.

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

4.4 Yielding

Observe that the freezing approach of Section 4.3 ends up reimplementing a lot of the behaviour of the freezing of ordinary commands, without adding much new behaviour. Furthermore, the term gets automatically unfrozen at the closest handler, which may not be what we want. 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

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

Figure 4.4: Unfreezing Computations Rule.

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

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

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

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;

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

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

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

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

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

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

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

4.5 Counting

The system described in Section 4.4 is slightly problematic; not only is it nondeterministic, but we can insert a yield whenever we want. If we spend too much time inserting and handling yield commands little other computation will be done.

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

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

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

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

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

R-HANDLE-COUNT

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

R-YIELD-CAN

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

R-YIELD-CAN'T

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

Figure 4.6: Yielding with Counting

use $+_c$, which will set the counter to be yield if the addition brings it over the threshold value.

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

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

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

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

Theorem 1 (Counting Implements Nondeterminism)

- *For any use m and counter c_y, if $m, c_y \rightsquigarrow_{\mathrm{u}} m', c_y'$ 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.

4.6 Handling

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

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

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

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.

These commands can 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, easily toggling which arguments should be interruptible by adding Yield to the corresponding interface.

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

4.7 Starvation

Consider the following program;

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

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

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

We would like `tree!` to print out `"A B C A B C A B C ..."`. However, when

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

$$\textsc{R-Handle}\quad \frac{\begin{array}{c} k = \min_i \{i \mid \exists \overline{\theta}.(r_{i,j} : \langle \Delta_j \rangle A_j \leftarrow t_j \dashv [\Sigma]\ \theta_j)_j\} \\ (r_{k,j} : \langle \Delta_j \rangle A_j \leftarrow t_j \dashv [\Sigma]\ \theta_j)_j \qquad \forall\ j \leq\ n\ .\ c_j \neq \mathsf{yield} \end{array}}{\mathcal{E}[\uparrow(\{((r_{i,j})_j \to n_i)_i\} : \{\overline{\langle\Delta\rangle A \to} \ [\Sigma]B\})\ \bar{t}];c_n,\ldots,c_1 \rightsquigarrow_{\mathrm{u}} \mathcal{E}[\uparrow((\overline{\theta}(n_k) : B))];c_n{+_c}1,\ldots,c_1{+_c}1}$$

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

$$\textsc{R-Yield-Can't}\quad \frac{\begin{array}{c} \neg(\mathcal{E} \text{ allows Yield}) \\ \forall\ j \leq\ (n-1)\ .\ c_j \neq \mathsf{yield} \qquad \mathcal{E}[m];c_n,\ldots,c_2,c(k) \ \rightsquigarrow_{\mathrm{u}}\ \mathcal{E}[m'];c_n,\ldots,c_2,c'_1 \end{array}}{\mathcal{E}[m];c_n,\ldots,c_2,\mathsf{yield} \rightsquigarrow_{\mathrm{u}} \mathcal{E}[m'];c_n,\ldots,c_2,\mathsf{yield}}$$

Figure 4.7: New counting, backwards ordered though

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

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

We have to update the semantics with the following rules to make this possible. Our transitions are now of the form $m, \overline{c_y} \rightsquigarrow_{\mathrm{u}} m', \overline{c_y}$. This list of counters is the list of counters *above* the current term in this tree of multihandlers. For instance, **`echo "A "`** will maintain the counter for **`sched`** directly above it; but **`echo "B "`** maintains the counter for both multihandlers above. The rules are shown in Figure 4.7. We label this system $\mathbb{F}_{\mathcal{T}}$. As before, we adopt the syntactic sugar that if transitions are not labelled with the counters then the counters remain unchanged.

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

our syntax tree.

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

TODO: Check Add counter????

Observe that this now means all threads scheduled in `tree` will get processor time. As `echo ``B''` evaluates the counter at both schedulers is incremented. They both pass over the threshold at the same time, and thus both counters are in yield. However, the lower scheduler is not allowed to insert a yield yet as the upper one blocks it. This is what the constraint $\forall\, j \leq (n-1)\,.\, c_j \neq \mathsf{yield}$ implements. Thus the higher scheduler yields first, and `echo ``A''` gets processor time. With this, we can express a liveness property of the new system.

Theorem 2 ($\mathbb{F}_{\mathcal{T}}$ Liveness) *Given a multihandler m with n threads to be scheduled, any thread handled by m receives processor time in $n * t_y$* local *reductions.*

A local reduction is a reduction that happens underneath the multihandler in question. In the function `tree` above, when `echo "A "` reduces it is local to the top-level `sched` handler but not the lower one.

This directly implies that we do not have thread starvation in $\mathbb{F}_{\mathcal{T}}$ as we do in $\mathbb{F}_{\mathcal{C}}$.

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 3 (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

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

Assume that

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

From $\mathcal{E}$ allows yield we know that $\mathsf{yield} \in \Xi_{|\bar{t}|}$ 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 4 (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

In this section we give a brief introduction to programming with asynchronous effects, and introduce the Frank library for doing so. Our design closely follows Æff (Ahman and Pretnar [2020]).

One can consider the traditional treatment of shallow effect handling as having three stages. First an operation op is invoked, with arguments V and continuation $\lambda x.M$. Then the handler for op — being the *implementation* of op — is evaluated until it returns some value V. Finally, the continuation of the caller is resumed by binding V to x in M.

What makes effect handling *synchronous* is that the operation call op *blocks* until the continuation M is resumed. This means that for *every* algebraic effect, the rest of the computation has to wait for the handler to be performed, even when the results are not immediately needed.

The *asynchronous* treatment of effect handling decouples these three stages; each of invoking an effect, evaluation of the handler, and resumption of the caller are separate. This permits the non-blocking invocation of effects; we can invoke an operation, continue with other work, then when (or even if) we need the result of the operation we can choose to block.

TODO: I'm not convinced by the use of "the rest of the computation"

Consider effect handling as a simple two threaded application; one thread is the caller, which invokes some operation op, and the other is the handler for op. In the synchronous treatment, once the caller invokes `op` the rest of the caller is blocked until the handler has finished handling `op`, at which point the caller restarts. In the asynchronous treatment, the caller can continue executing; once it needs the result of the operation it can then choose to block.

Asynchronous effects are used for writing multithreaded programs. A single thread might handle some operations and also perform other ones, which themselves are handled by other threads. In the rest of this section we explain this behaviour by example, and introduce our library for programming with asynchronous effects in Frank.

5.1 Communication

Consider a program $\mathcal{F}$ which lets the user scroll through an seemingly infinite feed of information (example due to Ahman and Pretnar). The program displays each item in the cache of data as the user scrolls; the program simulates being infinite by making a request for another cache of data whenever the user is nearly at the end of the current cache. In this way, the user never notices the feed pausing to download more data.

Implemented with asynchronous effects, $\mathcal{F}$ would send *signals*, being requests that some abstract operation is performed, and *interrupts*; these are requests that $\mathcal{F}$ itself performs an operation. For instance, $\mathcal{F}$ would send signals such as `display d` to ask that data `d` is displayed to the user, or `request`, to ask for a new cache of data from the server. Similarly, $\mathcal{F}$ would receive interrupts such as `nextItem`, being a request that the client loads the next piece of data, or `response x`, being the server responding to a request for more data.

Using asynchronous effects, threads communicate with one another using *signals*, which are analogous to effect invocations. For instance, $\mathcal{F}$ would use a `request` signal to ask for more data from the server. Signals sent from other threads become *interrupts* from the receiver's point of view; for instance, $\mathcal{F}$ would receive `response` interrupts from the server, containing the new cache of data.

TODO: Maybe say "operation calls are further split into signals interrupts?"

Despite this example of fairly conventional, we remark that each signal need not have a corresponding interrupt response and vice versa. For instance, in Section 6.1 we implement a pre-emptive scheduler, which sends `stop` and `go` signals without expecting an interrupt as a response.

TODO: Talk about potentially having the system intercept signals?

TODO: Maybe highlight that there's not a material difference between sigs and interrupts and that it's just a naming thing

TODO: Maybe we don't need display?

We define *interrupt handlers* to dictate how to act when an interrupt is received. An interrupt handler is a function of type `S -> Maybe {R}`, where `S` is a sum type

made up of the possible interrupts that can be received. The return type of the handler is `Maybe {R}` as we can choose *not* to handle the interrupt by returning `nothing`; this could be because it is the wrong type of signal, or if some other condition regarding the interrupt is not fulfilled[1]. An example of an interrupt handler is `boringFeed`;

```
data Feed = nextItem | request Int
          | response (List Int) | display Int

boringFeed : {Feed -> Maybe {[Console] Unit}}
boringFeed nextItem = just {print "10"}
boringFeed _ = nothing
```

When a thread installs the interrupt handler `boringFeed`, it prints out `10` the first time it receives a `nextItem` interrupt.

The set of different signals and interrupts a computation can send and receive is encoded in a sum type where each different constructor is a different signal name, with extra parameters encoding the payload of the signal. The datatype `Feed` above is an example of such a type.

From henceforth we also refer to interrupt handlers as *promises*; an interrupt handler for `op` once installed is a *promise* to perform the given action once an `op` interrupt is received.

TODO: Maybe get rid of this part.
TODO: Talk about system?

5.2 An Interface for Asynchronous Effects

To make our ideas more concrete, we introduce the Frank interface used for programming with asynchronous effects. First of all we introduce the datatype used to track the state of an installed promise, `Prom`.

```
data Prom X = prom (Ref (PStatus X))
data PStatus X = waiting | done X | resume {X -> Unit}
```

A value of type `Prom X` is a reference to a value of type `PStatus X`. It is stored as a reference as we have to write to this cell from two locations; the interrupt handler itself updates the cell when it has been fulfilled, and the handler for `await` commands also has to access the cell when the promise is awaited.

[1] An interrupt handler which inspects the body of the interrupt before executing is called a *guarded* interrupt handler. We see an example of guarded interrupt handling in Section 6.1.

TODO: Justify reference better?

A promise has three possible staes, each a different constructor for `PStatus`. The first, `waiting`, expresses that the promise has not yet been fulfilled. The second, `done x`, expresses that the promise has completed and resulted in a value, `x`. The third option, `resume cont`, is used when a promise is awaited but has not yet completed; in this case, the handler for `await` writes the continuation of the caller to `resume`. The interrupt handler then automatically resumes this once it is fulfilled.

TODO: Talk about caller not getting access to the Pid cell

```
interface Promise S =
    promise R : {S -> Maybe {[Promise S, RefState, Yield] R}}
             -> Prom R [Promise S, RefState, Yield]
  | signal : S -> Unit
  | await R : Prom R [Promise S, RefState, Yield] -> R
```

The entire `Promise` interface is polymorphic in the type of *signals* that threads can perform. This will be a datatype such as `Feed`, as discussed earlier. The commands themselves are polymorphic in the result types.

The `promise` command is used to install an interrupt handler; it takes an interrupt handler and returns a `Prom R` value. The interrupt handler can perform further `Promise S` effects, and must also have access to the `RefState` interface; we show why later. The `Yield` interface is also present so that interrupt handlers are themselves pre-emptible when executed.

The `signal` command takes a value of the `S` type and returns `unit`. When handling `signal sig`, all other threads are informed that a thread has invoked the signal `sig`. No matter what they were previously doing, all installed interrupt handlers now have to handle this signal. This process is the namesake of interrupts; the other threads are interrupted with this signal.

For every thread, all installed interrupt handlers are triggered with the incoming interrupt. Those that reduce to `nothing` are reinstalled; those that reduce to `just thunk` immediately perform the `thunk` before continuing with the interrupted computation.

Finally, the `await` command takes a `Prom R` value and returns a value of type `R`. At this point, we inspect the promise state as stored in reference. If the promise is still `waiting`, we take the continuation `cont` offered up by `await` and store it in the cell as `resume cont`. If the promise is `done` we immediately resume the continuation with the stored value. At this point the cell should not have a continuation in it, as it's not possible for multiple threads to await a single promise. As such, we just safely exit.

When a promise is fulfilled, it looks inside its associated `Prom` cell. If the status is just `waiting`, the promise just writes the returned value to the cell as `done x`. If there is a resumption in the cell, the promise immediately resumes it. There should not already be a `done x` value in the cell, as only the given promise and the handler for the promise interface should have access to it.

TODO: Better thing for what we do if there's a continuation in there.

5.3 In Action

Let's revisit the infinitely scrolling feed example from earlier, and consider the client thread, $\mathcal{F}$. The bulk of the client is an interrupt handler for `nextItem` messages. The body of this handler will display the next datum and reinstall the interrupt handler, as well as perform any requests for extra data. The type signature of the body of our handler will be

```
onNext : {List Int -> Maybe (Prom (List Int) [InThread])
          -> [InThread] Unit}
```

where `[InThread]` is an *interface alias* for `[Promise Feed [Console], Console, RefState, Yield]`. The first argument to `onNext` is the currently stored cache of data. The second argument is a `Prom` cell which may not be present; this stores the promise that waits for a response from the server when a request for extra data is made.

We use a helper function, `displayRestart`, to do some tasks that happen every time `onNext` is executed;

```
displayRestart : {List Int -> Maybe (Prom (List Int) [InThread])
              -> [InThread] Unit}
displayRestart cache p =
    signal (display (head cache));
    let cache = pop cache in
    promise { nextItem -> just { onNext cache p } | _ -> nothing };
    unit
```

We simply `display` the first element of the cache, and then install an interrupt handler for `nextItem` interrupts that reinstalls `onNext`, with the top item removed from the cache.

For the sake of simplicity we assume that the cache size is fixed to 10 items. Then whenever we have 3 or less items in the cache, and another request is not already in progress, we want to issue a new request for data.

```
onNext xs nothing = if (len xs == 3)
    { let resp = promise {(response x) -> just {x} | _ -> nothing}
       in
      signal (newData (last xs));
      displayRestart xs (just resp) }
    { displayRestart xs nothing }
```

Observe that if the length of the cache is 3 we first install an interrupt handler for **response** interrupts, and then issue a **newData** signal; we know that the server will respond to the **newData** interrupt it receives with a **response** message. As mentioned before, not every signal sent has a corresponding interrupt that will later be received; for instance, the **display** signal is sent without requiring an interrupt to respond, and the **nextItem** interrupt is received without the client sending a signal to cause it to come in.

Once the request for new data has been issued, we reinvoke **onNext** (via **displayRestart**), but this time carrying the promise. This leads us to the other branch of **onNext**;

```
onNextList cache (just p) =
    if (len cache == 0)
         { let newCache = await p in
           displayRestart newCache nothing }
         { displayRestart cache (just p) }
```

Here we check if we are at the end of the current cache. If we are, we await the promise, binding **newCache** to the result. Once the promise **p** is fulfilled we proceed as normal with the cache returned from the **resp** promise.

The client can then use this promise by installing it in the same way as in **displayRestart** ; they may want to install another interrupt handler, to listen for other messages from the user interface.

5.4 Modelling Asynchrony

Whilst it is convenient to describe our implementation as truly asynchronous, it is not in practise.

TODO: Better opening paragraph

We use the system as described in Chapter 4 to automatically force threads to yield after a certain amount of time passes. This gives the benefit of not having to worry about manually inserting yields, as previously discussed.

We use a scheduler similar to as descirbed in Section **??**; rather than a multihandler, which has a fixed number of threads to be handled, we use the `Threads` datatype as discussed earlier to store our threads in. When one yields, we take the next one and start executing that.

Earlier we mentioned that when a thread sends a signal, all other threads immediately perform the body of any fulfilled promises. This is not true in practise; all other threads are notified that they should perform the bodies of the fulfilled promises, but they only do so when the performing thread gets processor time. This is a more structured approach than that taken by Æff, where the body of a fulfilled promise may be performed instantly.

Chapter 6

Examples

Many languages which support async-await — such as C and Javascript — have the behaviour built-in. First the compiler is changed to add new syntax, then the compiler is changed to add new type-checking, then we have to implement the semantics; even worse, we have to do this when we want another asynchronous primitive, such as futures (asynchronous post-processing of results).

We show that with our promise library we can implement all of these common asynchronous primitives within the language itself.

6.1 Pre-emptive Scheduling

Whilst we have already shown how to pre-emptively schedule several threads in Section 4.6, we might want to have a more robust way of doing this; the multihandler strategy is fixed in a left-to-right evaluation order. In this method, we can just have a single source sending out `stop` and `go` messages, implementing a potentially more sophisticated scheduling strategy than mere round-robin.

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

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

```
onStop : {Int -> [Promise] 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] Unit}}
stopPromise id (stop n) = if (n == id)
                            { just { onStop id } }
                            { nothing }
stopPromise id _ = nothing
```

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

```
goPromise : {Int -> Sig -> Maybe {[Promise] 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**.

To make a thread pre-emptible, we simply install the stop interrupt handler before the main body of the computation. The computation being scheduled is then entirely unaware it is being scheduled, as desired.

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

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

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

One disadvantage to our approach is that an adversarial thread could just send **stop** and **go** signals of its own, overriding the scheduler. Using session types (Honda et al. [1998]) to restrict inter-thread communication would be able to solve this problem.

6.2 Futures

Our developed asynchronous effects system is expressive enough to implement the asychronous post-processing of results, or *futures*, on top of what we already have. Previously these have had to be implemented as a separate language feature (Schwinghammer [2002]).

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

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

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

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

6.3 Async-Await with Workers

Our asynchronous effects system can express the familiar async-await abstraction. This had previously been implemented in Frank by forking new threads, then to be handled by a co-operative scheduler like in Section 2.3. We realise it by using a controller thread, which will send tasks to one of a set of n worker threads. When these

worker threads are not working, they are instantly skipped; hence there is not much inefficiency associated with having extra idle workers.

We first show how the caller communicates with the controller.

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

async : {[{String} -> Ref Int -> [Promise] Pid String [Promise]
async proc callCounter =
    let callNo = read callCounter in
    let waiter = {s -> resultWaiter callNo s} in
    signal (call proc callNo);
    write callCounter (callNo + 1);
    waiter
```

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

The controller installs an interrupt handler to react to **`async`** interrupts. The thread tracks which threads have a task running; if there is a free worker it then performs a **`workIn`** message, containing the computation and the ID of the worker who should perform it. The controller then installs a promise to update the active status of the corresponding worker once a **`result`** interrupt is received.

TODO: Introduce what all of the signals are ????

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

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

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

feature of the asynchronous effects system.

6.4 Cancelling Tasks

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

```
interface Cancel = cancel : Unit

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

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

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

workProm : {Int -> Sig -> Maybe {[Promise] Unit}}
workProm wid (workIn p wid' callId) =
    if (wid == wid')
       { just {promise (prom {s -> canceller wid callId s};
               let res = p! in
               signal (result res callId);
               worker wid; unit}}
       { nothing }
workProm _ _ = nothing
```

We have to modify the handler for the Promise effect for this. Recall that when we install a promise, we convert it to a form that takes composes the body of the promise with the interrupted computation and rehandles the `Promise` effects that this might perform. We need to then wrap this in a handler for Cancel effects again; this is because user-level promises could perform Cancel effects, and we want to cancel the

whole computation, not just the body of the promise.

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

The realisation of cancellable function calls in Æff (Ahman and Pretnar [2020]) was to start awaiting a new promise that will never be fulfilled. 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 our system is that we have to modify the handler code for promises, even though cancellation of calls and promises should be orthogonal.

6.5 Interleaving

With the Cancel effect, we can also define the useful `interleave` combinator, in the spirit of Koka's interleave operator Leijen [2017a]. This lets us issue two tasks on two different threads, using the `call` signal as defined in Section 6.3. We then install an interrupt handler which listens for `result` interrupt; if a `result` interrupt corresponds to one of the installed threads we cancel the other thread using `cancelCall` and return the received result.

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

Observe the similarity of `interleave` and `async`. By taking asynchronous programming structures out of library code and into the programmer's hands, we hope that programmers will be able to more easily produce their own, useful tools.

Chapter 7

Conclusion

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

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

TODO: Add more to end of paragraph

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

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

Bibliography

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

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, 84(1):108–123, 2015.

Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 133–144, 2013.

Lukas Convent. *Enhancing a modular effectful programming language*. PhD thesis, MSc thesis, School of Informatics, The University of Edinburgh, 2017.

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

Stephen Dolan, Leo White, and Anil Madhavapeddy. Multicore ocaml. In *OCaml Workshop*, volume 2, 2014.

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

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

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

Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, pages 122–138. Springer, 1998.

Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *ACM SIGPLAN Notices*, 48(9):145–158, 2013.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices*, 48(12):59–70, 2013.

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

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

Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 486–499, 2017b.

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

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

Jan Schwinghammer. A concurrent lambda-calculus with promises and futures. Master's thesis, 2002.

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