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 to Sam Lindley for giving me the opportunity to write this dissertation, and for his feedback and support throughout, right up to the end.

Thanks to everyone who made the past year so much fun. Thanks to my parents and grandparents for all of the support throughout the whole process.

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 (Lindley et al. [2017], Convent et al. [2020]), a functional programming designed with effect handlers at its core. We follow the design of Æff (Ahman and Pretnar [2020]), a small programming language designed around asynchronous effects but supporting little else. We show how 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. Effect handlers have already shown 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 backend. 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.

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 introduce the asynchronous effects abstraction and explain how it is implemented 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.

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.

Thunks *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.

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.

TODO: Talk about ABILITIES

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 !)`.

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`.

A similar thing happens in interface declarations; the interface `Co` below desugars to `CoVerbose`:

```
interface Co X             = fork : {[Co X] X} -> Unit
interface CoVerbose X [ε] = fork : {[ε | Co X [ε|]] X} -> Unit
```

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.

2.1 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. 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.

2.1.1 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.

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

```
schedule : {<Yield> Unit -> <Yield> Unit -> Unit}
schedule <yield -> m> <yield -> n> = schedule (m unit) (n unit)
schedule <yield -> m> <n> = schedule (m unit) n!
schedule <m> <yield -> n> = schedule m! (n unit)
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.

2.1.2 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 operation `fork` takes a suspended computation that can perform further `Co` effects and returns unit once handled. An example program using this interface is `forker`;

```
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 the forked thread later, 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.

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. Later in this thesis we refer to the system presented in this chapter as $\mathbb{F}$.

(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)	$\Sigma ::= \sigma \,\vert\, \Xi$
(extensions)	$\Xi ::= \iota \mid \Xi, I\,\overline{R}$
(adaptors)	$\Theta ::= \iota \mid \Theta, I(S \to S')$
(adjustments)	$\Delta ::= \Theta \,\vert\, \Xi$
(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

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

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

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

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

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

Figure 3.3: Runtime Syntax

values or *frozen commands*, $\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil$. Frozen commands are used to capture a 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-

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

R-HANDLE

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

R-ASCRIBE-USE

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

R-ASCRIBE-CONS

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

R-LET

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

R-LETREC

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

R-ADAPT

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

R-FREEZE-COMM

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

R-FREEZE-FRAME-USE

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

R-FREEZE-FRAME-CONS

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

R-LIFT-UU

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

R-LIFT-UC

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

R-LIFT-CU

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

R-LIFT-CC

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

Figure 3.4: Operational Semantics

FRAME-CONS show how the rest of the context becomes frozen. These two rules rely on the predicate $\mathcal{E}$ handles c. This is true if the context does indeed handle the command c; i.e. it is a context of the form $u\ (\overline{t}, [\ ], \overline{u'})$ 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. The relation $p : A \leftarrow w \dashv \theta$ states that a value pattern p of type A matches normal form w yielding substitution θ. B-VAR states that any pattern w matches a value x, whilst B-DATA states that a constructor pattern $k\overline{w}$ matches a construction term $k\overline{p}$ if each subpattern p_i matches an argument to the construction w_i.

The rules regarding $r : T \leftarrow t \dashv[\Sigma]\ \theta$ are more interesting. B-VALUE defers com-

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

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

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

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

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

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

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

Figure 3.5: Pattern Binding

putation pattern matching onto value pattern matching. B-REQUEST expresses that a computation pattern $\langle c\ \overline{p} \to z\rangle$ matches a frozen computation $\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil$ if command c is handled by the evaluation context $\mathcal{E}$, and if the arguments to the command each match a subpattern in the computation pattern.

The catchall pattern **<x>** matches any value and any command that is handled by the current evaluation context; B-CATCHALL-VALUE and B-CATCHALL-REQUEST express this. Observe that B-CATCHALL-REQUEST has the same constraints as B-REQUEST; the computation pattern only matches a command if it could otherwise be handled.

Chapter 4

Pre-emptive Concurrency

4.1 Motivation

Our scheduler in Section 2.1 relies on threads manually yielding. This is fine for simple examples, but when working with more complex programs this is inconvenient; the programmer must insert yields with a consistent frequency, so as to avoid process starvation. Furthermore, if we use external or library functions these will not hold yields, so will be uninterruptible. It would be simpler and fairer to just use some automatic way of yielding.

TODO: Redo above

Consider the two programs below;

```
interface Stop = stop : Unit
interface Go = go : Unit

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

runner : {Int -> [Console] Unit}
runner x = printInt x; runner (x + 1)
```

We want a multihandler that uses the `stop` and `go` commands from `controller` to control the execution of `runner`. The desired console output is `1 2 3 ... (n-1)n stop! (n + 1)...`, running infinitely.

The problem as it stands is that there is no way for `runner` to be suspended whilst it is running; it will just infinitely run, never giving control to the handler or to `controller`
.

As an example, we show how we can approximate the desired behaviour using the familiar `Yield` interface from Section 2.1.1.

```
runner : {Int -> [Console] Unit}
runner x = printInt x; yield!: runner (x + 1)

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 0)controller! nothing` then prints out `1 stop 2 stop 3 stop ....` This is due to the same synchronisation behaviour that we saw in Section 2.1.1; `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.

In this way we use yield commands to split up our computations and let processing time be given to other computations. The closest handler for yield operations then gets access to the continuation of `runner` and can choose how to handle it.

We are, however, still operating co-operatively; the programmer has to manually insert yield commands. Furthermore, in this case we yield far too often; it would be more efficient to have a consistent, yet longer, period in between each yield command. 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 rule 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

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

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. The key constraint that has been removed is $c : \forall \overline{Z}.\overline{B \to} B' \in \Xi$, which requires that the frozen command must be present in the argument extension Ξ. The constraint $\mathcal{E}$ poisedfor c just states that the evaluation context containing the frozen command will handle c; we do away with this, as we do not necessarily want to handle the command here.

This lets us change **`runner`** back to its original form, and update **`suspend`** like so;

```
runner : {Int -> [Console] Unit}
runner x = printInt x; runner (x + 1)

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 0)controller! nothing`**, the **`suspend`** handler can match the catchall pattern **`<r>`** against the **`print`** commands in **`runner`**. This prints out `1 stop! 2 stop! 3 stop! ...` as before.

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 — such as `1 + 1; 1 + 1; 1 + 1` — we would be unable to interrupt it.

As such, we make a more significant change to the semantics of Frank. We adapt the syntax so that *any* term may become frozen, and not just commands; this is reflected

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

Figure 4.2: Runtime Syntax, Updated with Freezing of Uses

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

R-FREEZE-USE
$$\frac{}{m \rightsquigarrow_{\mathrm{u}} \lceil m \rceil}$$

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

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: Freezing Uses

in Figure 4.2. In Figure 4.3 we see additional rules for freezing arbitrary *uses* and the surrounding computations. We can freeze arbitrary *constructions* in an identical fashion, substituting m for n. These rules rely on an extra predicate $\mathcal{F}$ not handler , which is true unless $\mathcal{F}$ is of the form $u\ (\overline{t},[\,],\overline{n})$. Frozen terms behave very much like frozen commands, freezing the entire computation up to the nearest handler. Finally, we supplement the pattern binding rules with the rule in Figure 4.4, which shows how a computation becomes unfrozen. A frozen computation $\lceil m \rceil$ can match against the catchall pattern $\langle x \rangle$; the suspended, thawed computation $\{m\}$ is then bound to x in the continuation.

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

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

Figure 4.4: Thawing Computations.

However, observe that the frozen term is automatically rehandled at the closest handler. This is problematic; we might have a handler for another effect, such as `State` , before the `suspend` handler. In this case, the handler below would automatically resume `runner` when it freezes; we would still have the same problem of starvation, as control would never rise to `suspend`. This problem would be solved if we had finer-grained control over when to resume a frozen computation, so we could choose to resume the frozen computation at `suspend` and not the lower-down handler.

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, severely limiting control over computations. 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.

Once again, the simple $\mathsf{Yield} = \mathsf{yield} : \mathsf{Unit}$ interface from Section 2.1 can be used here. Whilst the interface itself sounds very boring, its use here comes from the fact it freezes the rest of the computation around it up until the next Yield handler.

Our new system is simple; whenever a term reduces underneath a handler for Yield effects, we insert an invocation of the yield command before the reduct. This is expressed formally in Figure 4.5. We refer to $\mathbb{F}$ as described in Chapter 3 supplemented with this rule as $\mathbb{F}_{\mathcal{ND}}$.

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

$$\textsc{R-Yield}\quad \frac{n \rightsquigarrow_{\mathrm{c}} n' \qquad \mathcal{F}\ \mathsf{allows\ yield}}{\mathcal{F}[n] \rightsquigarrow_{\mathrm{u}} \mathcal{F}[\mathsf{yield!}; n]}$$

Figure 4.5: Inserting Yields

Note that R-YIELD-EF relies on the predicate $\mathcal{F}$ allows c. For any frame apart from argument frames (i.e. $u(\bar{t}, [\,], \overline{n})$), $\mathcal{F}$ allows $c = \mathsf{false}$. In this case, it is defined as follows;

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

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

Informally, $\mathcal{F}$ allows c is true when $\mathcal{F}$ 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 are pre-emptible. The programmer can easily toggle this by labelling the pre-emptible threads with yield in the ability. This is one improvement over the system of Section 4.3; previously every thread was interruptible. Another benefit is that we define fewer new rules and constructs. We also benefit from the choice of when to resume a computation; in the previous system computations were automatically unfrozen at the nearest handler, but this problem is fixed in $\mathbb{F}_{\mathcal{ND}}$. Finally, we can write custom handlers for yield commands, whilst the unfreezing rules in Figure 4.4 was fixed at just restarting the continuation.

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 being printed to the console.

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

R-HANDLE-COUNT

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

R-YIELD-CAN

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

R-YIELD-CAN'T

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

Figure 4.6: Yielding with Counting

4.5 Counting

The system described in Section 4.4 is slightly problematic; 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. Furthermore, it is non-deterministic; we often have the choice of either yielding or reducing as normal. We need a way to decide whether or not to yield.

To combat this we supplement the operational semantics with a counter c. This counter has two states; it could either be counting up, which is the form $\mathsf{count}(n)$ 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 $\oplus$. This is simply defined as

$$x\oplus y = \begin{cases} \mathsf{count}(x+y) & \text{if } x+y \leq \mathsf{t} \\ \mathsf{yield} & \text{otherwise} \end{cases}$$

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

The transitions in our operational semantics now are of the form $m;\mathsf{c} \rightsquigarrow_{\mathrm{u}} m';\mathsf{c}'$. 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 $\mathbb{F}$ extended with the rules in Figure **??** as $\mathbb{F}_C$.

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

R-YIELD-CAN and R-YIELD-CAN'T dictate what to do if the counter is in the yield state. 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 were of the form count(k), then we make that transition, still maintaining the yield signal.

Note that we have a family of 4 R-YIELD-CAN'T rules, for each pair use / construction inside the evaluation context, which can be a use or a constructio, in a similar way to the R-LIFT rules in Chapter 3. We omit these for brevity.

All of the other rules from Figure 3.4 are then implicitly converted to $m;k \rightsquigarrow_{\mathrm{u}} m';k$; that is to say they may reduce at any point regardless of the state of the counter, but they do *not* change the value of the counter.

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 4.6 are now deterministic; for any term and counter pair, there is only one possible reduction we can make. This is helpful for the sake of implementation; it is always clear which reduction to make at any point. 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 implement the nondeterministic system. We have implemented the counting behaviour of $\mathbb{F}_{\mathcal{C}}$ into the Frank interpreter.

Theorem 1 ($\mathbb{F}_{\mathcal{C}}$ Implements $\mathbb{F}_{\mathcal{ND}}$.)**.**

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

Proof. If we simply ignore the counters it's clear that any time we insert a yield command in $\mathbb{F}_{\mathcal{C}}$, it is valid to also do so on $\mathbb{F}_{\mathcal{ND}}$. $\square$

In Section 4.7 we see a different approach, rather than a global counter, which also implements the nondeterministic semantics.

4.6 Handling

Observe that we can now use the same **suspend** handler from Section 4.1, without having to manually insert yield commands in **runner**. The following code will then give the desired output, of a series of numbers printing interspersed evenly with **stop !**;

```
runner : {Int -> [Console] Unit}
runner x = printInt x; runner (x + 1)

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 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 -> r> <yield -> c> p = suspend (r unit) (c unit) p
suspend <yield -> r> <c>          p = suspend (r unit) c! p
suspend <r>          <yield -> c> p = suspend r! (c unit) p
suspend <r>          <c>          p = suspend r! c! p
```

These let yield commands synchronise with each other, achieving fair scheduling, as discussed in Section 2.1. It is 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 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 to resume and rehandle all n yield commands. After this we try to 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 them. It is important to insert the automatically resuming patterns *after* the rest of the patterns, as the multihandler may want to handle yield commands some other way; we do not want to interfere with this. This means we can program in a simpler, 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 very little overhead to the programmer.

4.7 Starvation

Consider the following program;

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

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

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

We would like `tree!` to print out `"A B C A B C A B C ..."`. However, when using $\mathbb{F}_C$ with automatic insertion of yield commands, 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).

What we need is for each multihandler to have its own counter, which is incremented every time an argument to the multihandler reduces. When an argument to a multihandler reduces when the counter is over the threshold, we insert a yield command in front of the reduct.

This system is expressed formally in Figure 4.7. Every handler — just being a

R-HANDLE-GC

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

ARG-INCREMENT

$$\frac{n \leadsto_{\mathrm{c}} n'}{\begin{array}{c}\uparrow(\{((r_{i,j})_j \to n_i)_i\}@c : \{\overline{\langle\Delta\rangle A \to}\ [\Sigma]B\})\ (\overline{t}, n, \overline{n}) \leadsto_{\mathrm{u}} \\ \uparrow(\{((r_{i,j})_j \to n_i)_i\}@(\mathsf{incOrReset}(c, \Delta_{|\overline{t}|})) : \{\overline{\langle\Delta\rangle A \to}\ [\Sigma]B\})\ (\overline{t}, \mathsf{maybeYield}(n', c, \Delta_{|\overline{t}|}), \overline{n})\end{array}}$$

Figure 4.7: Updated Counting Rules

collection of pattern matching rules $\{((r_{i,j})_j \to n_i)_i\}$ — is implicitly given a counter c initialised at $\mathsf{count}(0)$. The first rule, R-HANDLE-GC[1], expresses that counters are removed when a handler is evaluated on fully-evaluated arguments. The second rule, ARG-INCREMENT, expresses that when an argument to a multihandler evaluates, we increment the counter; if the counter is above the yielding threshold we insert a yield command before the argument and reset the counter. ARG-INCREMENT relies on two auxilary functions, defined as;

$$\mathsf{incOrReset}(\mathsf{c}, \Delta = \Theta \mid \Xi) = \begin{cases} \mathsf{c} & \text{if } \mathsf{Yield} \notin \Xi \\ \mathsf{c}\oplus 1 & \text{if } \mathsf{Yield} \in \Xi \text{ and } \mathsf{c} \neq \mathsf{yield} \\ \mathsf{count}(0) & \text{if } \mathsf{Yield} \in \Xi \text{ and } \mathsf{c} = \mathsf{yield} \end{cases}$$

$$\mathsf{maybeYield}(n, \mathsf{c}, \Delta = \Theta \mid \Xi) = \begin{cases} \mathsf{yield!}; n & \text{if } \mathsf{Yield} \in \Xi \text{ and } \mathsf{c} = \mathsf{yield} \\ n & \text{otherwise} \end{cases}$$

We denote this system of $\mathbb{F}$ equipped with the rules in Figure 4.7 as $\mathbb{F}_{\mathcal{T}}$.

Theorem 2 ($\mathbb{F}_{\mathcal{T}}$ Implements $\mathbb{F}_{\mathcal{ND}}$.)**.**

- *For any use m if m* $\leadsto_{\mathrm{u}}$ m' *in* $\mathbb{F}_{\mathcal{T}}$ *then m* $\leadsto_{\mathrm{u}}$ m' *in* $\mathbb{F}_{\mathcal{ND}}$.
- *For any construction n, if n* $\leadsto_{\mathrm{u}}$ n' *in* $\mathbb{F}_{\mathcal{T}}$ *then n* $\leadsto_{\mathrm{u}}$ n' *in* $\mathbb{F}_{\mathcal{ND}}$.

Proof. We can see that this holds by just erasing the counters from the multihandlers; all transitions would be permitted in $\mathbb{F}_{\mathcal{ND}}$. □

We walk through an example evaluation of **`tree`** in $\mathbb{F}_{\mathcal{T}}$. First, **`echo "A "`** reduces, increasing the counter at the upper **`sched`** handler. Once this counter passes

[1]Where GC stands for Garbage Collector.

the threshold, we insert a yield before `echo "A "`; we now start evaluating the other branch, `sched (echo "B ")(echo "C ")`. While `echo "B "` reduces we increment the counter at both `sched` handlers. Both counters then pass the threshold at the same time. At this point the system can either choose to insert a yield at either of the two `sched` handlers; let's consider it chooses to insert one at the upper handler. Then `echo "A "` is evaluated again as before. Once we resume computing `sched (echo "B ") (echo "C ")` the counter state is *maintained*, so we immediately yield and evaluate `echo "C "` and continue.

In our implementation of $\mathbb{F}_{\mathcal{T}}$, we avoid the nondeterminism caused by multiple multihandler trying to yield by always choosing the lowest multihandler.

This system does not let threads starve; eventually, any thread gets processor time. However, if we have a lot of deeply-nested handlers, a thread might have to wait a long time to get processor time. It would be good to have a system where we can express a bound on the amount of time that will pass before a thread gets processor time; this remains as future work.

TODO: Rewrite above.

4.8 Soundness

We now state the soundness properties for our systems, as well as the subject reduction theorem needed for each soundness proof.

Theorem 3 (Subject Reduction for $\mathbb{F}_{\mathcal{ND}}$)**.**

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

Identical theorems hold for each of $\mathbb{F}_{\mathcal{C}}$ and $\mathbb{F}_{\mathcal{T}}$. Previous work (Convent et al. [2020]) has shown that subject reduction holds for $\mathbb{F}$; as such the proofs for each of our new systems amount to just showing the new reduction rules preserve types. Indeed, even this just amounts to showing that inserting a yield command before a term does not change the overall type of a term. Proofs of subject reduction for $\mathbb{F}_{\mathcal{ND}}$, $\mathbb{F}_{\mathcal{C}}$ and $\mathbb{F}_{\mathcal{T}}$ can be found in Section B.1.

Theorem 4 (Type Soundness for $\mathbb{F}_{\mathcal{ND}}$)**.**

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

Again, we have identical theorems for $\mathbb{F}_{\mathcal{C}}$ and $\mathbb{F}_{\mathcal{T}}$. The proof of type soundness proceeds by induction on $\cdot;\cdot\,[\Sigma]\!\vdash m \Rightarrow A$ and $\cdot;\cdot\,[\Sigma]\!\vdash n : A$. None of our extensions involve new typing judgements, so we do not add any new cases to the proof of type soundness for $\mathbb{F}$(Convent et al. [2020]). Our main obligation is to show that systems equipped with a counter never get stuck in a state where they cannot reduce due to the counter blocking; this is shown in Section B.2.

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"

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. This program could be run in parallel with a user interface controller and a server, among others.

The client thread $\mathcal{F}$ would then interact with these other threads by sending *signals* and receiving *interrupts*. One can imagine these as a further division of operation calls; a signal is the sender requesting an operation be performed, and an interrupt is the corresponding incoming request that the other threads receive. For instance, $\mathcal{F}$ would send a `request` signal to ask the server to send a new cache of data; $\mathcal{F}$ owuld then receive a `response x` interrupt, where `x` is the new data from the server.

Despite this example, we remark that signals do not require a corresponding interrupt as response and vice versa. For instance, $\mathcal{F}$ would perform `display d` signals, as requests to the UI controller to display data `d`; the client then doesn't need a response from the UI controller. Similarly, $\mathcal{F}$ receives `nextItem` interrupts whenever the user requests to see a new item.

TODO: Say something about "dynamic behaviour" or whatever

TODO: Talk about potentially having the system intercept signals?

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; an example is the `Feed` type defined below. 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"}
```

[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.

```
boringFeed _ = nothing
```

The interrupt handler `boringFeed` prints out `10` on receipt of a `nextItem` interrupt; if it receives any other interrupt it does nothing.

A thread will then *install* an interrupt handler to use it. Once installed, an interrupt handler is then informed of every interrupt that the installing thread receives. We formalise what exactly happens in the next section.

TODO: Fix this?

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. A promise has three possible states, 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. Ideally, `Pid` should be an abstract type; the programmer should not be able to directly look inside an installed `Pid`. The only way the programmer should get the value out of a `Pid` is by awaiting the promise.

```
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. We can also parametrise the `Promise` interface by effects that the interrupt handlers can perform. For instance, if using the `Promise S [Console]` interface interrupt handlers also perfom `Console` effects and further `Promise S [Console]` effects. This is due to implicit effect polymorphism as discussed in Chapter 2. A stack of installed interrupt handlers is kept for each thread.

The `signal` command takes a value of the `S` type and returns `unit`. When handling `signal sig`, all other threads are interrupted; they stop whatever they were doing, and all installed interrupt handlers now have to handle this signal. We go through each interrupt handler `ih` in the stack. Recall that an interrupt handler is just a function of type `S -> Maybe {R}`. Thus we simply apply `ih` to the interrupt `sig`. If `(ih sig)` $\rightsquigarrow$ `nothing` we leave `ih` on the stack and look at the next interrupt handler. If `(ih sig)` $\rightsquigarrow$ `(just thk)`, the interrupted thread immediately performs the thunk `thk` before continuing with the interrupted computation. In this case, `ih` is removed from the stack. We say that a promise which reduces to a `(just thk)` is *fulfilled*.

TODO: talk about how AEff non-confluent, etc

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 automatically 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.

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 practice.

We use the system as described in Chapter 4 to automatically force threads to yield after a certain amount of time passes. To handle these yield commands, we use a scheduler similar to as described in Section 2.1.2; 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 stricter approach than that taken by Æff, where an interrupted computation may still reduce before the interrupt handlers process the interrupt. However, this is a testament to the true asynchronous behaviour of Æff; this is only approximated in Frank. It remains as future work to mimic the full asynchronous behaviour of Æff.

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.

For simplicity's sake, we just display a version with only one thread, however this approach can easily be generalised to pre-empt multiple threads by adding ID fields to the **`stop`** and **`go`** signals and using guarded interrupt handlers for **`goPromise`** and **`stopPromise`**.

```
data Schedule = stop | go

goPromise : {Sig -> Maybe {[Promise Schedule] Unit}}
goPromise go = just {unit}
goPromise _ = nothing
```

```
stopPromise : {Sig -> Maybe {[Promise Schedule] Unit}}
stopPromise stop = just {await (promise goPromise);
                         promise stopPromise; unit}
stopPromise _ = nothing

preempt : {{X} -> [Promise Schedule] X}
preempt comp = promise stopPromise; comp!
```

We can easily make a computation pre-emptible by just installing `stopPromise` before the main body, as in `preempt`.

Once a preemptible computation receives a `stop` interrupt, it installs `goPromise` and immediately awaits it. This blocks the rest of the computation from executing until a `go` interrupt is received. When such a signal does come in, `goPromise` is fulfilled; the body of the interrupt handler does nothing, but it unblocks the rest of the thread's computation. At this point, the rest of the body of `stopPromise` is also unblocked, so another `stopPromise` is installed.

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 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 asynchronous effects system is expressive enough to implement the asychronous post-processing of results, or *futures*. 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, rather perform this action asynchronously, when the promise is complete.

```
data Fut = newData (List Int) | result Int

futureList : {Pid R [Promise Fut] -> {R -> [Promise Fut] Z} -> Sig
    -> Maybe {[Promise Fut] Z}}
```

```
futureList p comp (newData _) = 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 { (newData xs) -> just {xs} | _ -> nothing} in
let prod = promise {s -> futureList recv product s} in
promise {s -> futureList prod {x -> signal (result 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 2.1. 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 use three types of signal here. The calling thread sends `call` messages, where the arguments are the computation to run and the call ID. The controller handles `call` interrupts and sends `work` interrupts; the arguments to this signal are the computation again, the call ID and the ID of the worker who is designated to run this task. Finally, the worker sends `result` signals when it has finished computing; arguments to this are the result and the call ID.

```
data Async R = result R Int | call {R} Int
             | work {R} Int Int

async : {[{String} -> Ref Int
    -> [Promise (Async String)] Pid String [Promise (Async String)]
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 a promise `resultWaiter` 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` just returns the `result` promise; as such, the `await` operation is just the built-in `await` operation.

The controller installs an interrupt handler to react to `call` interrupts. The thread tracks which threads have a task running; if there is a free worker it then sends a `work` signal, 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 from the worker.

Workers listen for a `work` 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 issued with async-await run on their own thread; we can use the `cancel` effect to throw away the continuation of the entire thread and wipe the slate clean. To make our worker threads cancellable, we change the interrupt handler for **work** interrupts to install the **canceller** promise before the worker starts running the task:

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

The **canceller** promise reinstalls the **worker** promise before cancelling, so that the thread can eventually run another task again after the current task is done. The controller also is modified to install an interrupt handler for **cancelCall** interrupts; this interrupt handler sets the corresponding worker's state to idle.

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 truly disappear.

However, we have to modify the handler for the Promise effect to correctly cancel threads. When we fulfill a promise, we take the result computation and compose this with the interrupted computation and rehandle this single computation with the promise handler. At this point, we also now handle **Cancel** effects, so that we remove the whole thread. This is a point for improvement; handling the **Cancel** effect and handling **Promise** effects are orthogonal, and should be treated separately.

6.4.1 Interleaving

With the Cancel effect, we can also define the useful **interleave** combinator (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.

Observe that **interleave** is just a slight modification of **async** as defined earlier. By taking asychronous programming structures out of library code and into the programmer's hands, we hope that programmers will be able to easily craft their own tools.

Chapter 7

Conclusion

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

Combining the pre-emptive concurrency of Chapter 4 with the promise library of Chapter 5 yields a direct and comfortable way to write programs making use of asynchronous effects.

In Chapter 6 we showed how our system can be used to implement common, useful structures for asynchronous programming. We can recreate the behaviour of Æff, the only other language with support for asynchronous effects, and show how asynchronous effects in the prescence of synchronous effects can be used to cancel calls. In a more holistic sense, we hope that one of the outcomes of this project and related work (Ahman and Pretnar [2020], Leijen [2017a], Dolan et al. [2017]) is taking the definition of asynchronous programming features away from the realm of low-level operating system schedulers and opaque web programming interfaces and offering them up to the user.

7.1 Limitations and Future Work

Type System 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.

Interactions between Synchronous and Asynchronous Effects Handling effects performed by interrupt handlers can be quite a challenge sometimes. Effects like `Console` and random number generation are fairly straightforward to handle, as we can let them pass up to the top-level and handle with a single handler. However, handling something like the `State` effect is trickier; we cannot use the same `State` handler to handle everything as this would share the state between all threads. Threading the `State` handler through the `Promise` handler is cumbersome; it would be better to leave the `Promise` handler to be.

Session Types for Communication Protocols 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.

A Higher Degree of Asynchrony There are several degrees of asynchrony possible with a system like ours. We restrict one thread to run at any given time, with fulfilled promises only evaluating when the corresponding thread gets processor time. Æff take the other approach; any thread can compute at any time. There is an in-between, where the bodies of interrupts may be evaluated out-of-turn. It would be interesting to further explore the differences between different models of asynchrony and their benefits and limitations.

Sophisticated Yielding Strategies .....

Implementations in Other Languages

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.

Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. *POPL*, 2017.

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

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

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

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

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

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

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

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

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

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

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

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

Figure A.1: Term Typing Rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure A.4: Well-Formedness Rules

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

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

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

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

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

Figure A.5: Pattern Matching Typing Rules

Appendix B

Extended Proofs

B.1 Subject Reduction

Theorem (Subject Reduction for $\mathbb{F}_{\mathcal{ND}}$)**.**

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

Proof. The proof proceeds by induction on the transitions $\rightsquigarrow_{\mathrm{u}}, \rightsquigarrow_{\mathrm{c}}$. We need only address the R-YIELD rule, as all other rules have previously been shown to preserve types (Convent et al. [2020]).

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

$$\mathcal{F}\,[\,] = \uparrow\!(v : \{\overline{\langle\Delta\rangle A \to} \; [\Sigma]B\})\ (\bar{t}, [\,], \overline{n})$$

where yield $\in \Xi_{|\bar{t}|}$ and $\Delta_{|\bar{t}|} = \Theta_{|\bar{t}|} \mid \Xi_{|\bar{t}|}$.

So assume that $\Phi;\Gamma\,[\Sigma]\!\vdash \mathcal{F}\,[n] : B$. Then by inversion on T-APP we have that $\Phi;\Gamma\,[\Sigma'_{|\bar{t}|}]\!\vdash n : A_{|\bar{t}|}$ and $\Sigma \vdash \Delta_{|\bar{t}|} \dashv \Sigma_{|\bar{t}|}$. It follows then that $\Phi;\Gamma\,[\Sigma'_{|\bar{t}|}]\!\vdash (\mathsf{yield!}; n) : A_{|\bar{t}|}$, and accordingly that $\Phi;\Gamma\,[\Sigma]\!\vdash \mathcal{F}\,[\mathsf{yield!}; n] : B$.

$\square$

Theorem (Subject Reduction for $\mathbb{F}_{C}$)**.**

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

- *If* $\Phi;\Gamma\,[\Sigma]\!\vdash n : A$ *and* $n; \mathsf{c} \leadsto_{\mathrm{c}} n'; \mathsf{c}'$ *then* $\Phi;\Gamma\,[\Sigma]\!\vdash n' : A$.

Proof. Again we look at each of the new rules added by $\mathbb{F}_{\mathcal{C}}$.

Case R-HANDLE-COUNT Follows from subject reduction for R-HANDLE, as the terms are unchanged between the two rules.

Case R-YIELD-CAN Identical to R-YIELD from above.

Case R-YIELD-CAN'T Assume that $\Phi;\Gamma\,[\Sigma]\!\vdash \mathcal{E}[m] \Rightarrow A$, and let mB. By the inversion we have that $m; \mathsf{count}(k) \leadsto_{\mathrm{u}} m'; \mathsf{count}(k')$; by subject reduction we have that mB. It follows that $\Phi;\Gamma\,[\Sigma]\!\vdash \mathcal{E}[m'] \Rightarrow A$.

□

Theorem (Subject Reduction for $\mathbb{F}_{\mathcal{T}}$).

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

Proof. Essentially identical to the above two proofs. R-HANDLE-GC just removes the counter and otherwise acts identically to R-HANDLE. ARG-INCREMENT might insert a yield, but only if it's type-safe in the same way as before. □

We could also argue for subject reduction by using the fact that $\mathbb{F}_{\mathcal{T}}$ and $\mathbb{F}_{\mathcal{C}}$ both implement $\mathbb{F}_{\mathcal{ND}}$, and $\mathbb{F}_{\mathcal{ND}}$ preserves types when reducing.

B.2 Type Soundness Proofs

Theorem 5 (Type Soundness for $\mathbb{F}_{\mathcal{ND}}$).

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

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

$\mathbb{F}_{\mathcal{ND}}$ does not much complicate the proof. We can insert a yield command at any point when evaluating an argument to a handler that handles yield commands; this is then obviously a normal form that respects $\Sigma.If the$yield $command is not inserted then soundness follow$

Theorem 6 (Type Soundness for $\mathbb{F}_{\mathcal{C}}$)**.**

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

Proof. Here the main obligation is to show that the use of yield does not potentially block a computation from reducing when it otherwise could, thus breaking type soundness. When the counter is in the yield state, the only type of term it effects is $\mathcal{E}[m]$. If the evaluation context is a handler where the ability at the hole permits yield operations, we insert a yield; this freezes the rest of the term around it, becoming a normal form. If the evaluation context does not permit yields but the term could otherwise reduce then it does so. $\square$

Theorem 7 (Type Soundness for $\mathbb{F}_{\mathcal{T}}$)**.**

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

Proof. This proof is essentially the same as the above, showing that computation is never blocked by a counter. $\square$