Asynchronous Effect Handling

Leo Poulson

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2020

Abstract

Abstract

Acknowledgements

thanks!

Table of Contents

Chapter 1

Programming in Frank

1.1 Effects and Effect Handling

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

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

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

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

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

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

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

```
  runState x            _ = x
```

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

What happens when we run `runState xplusplus! 0`?

1.2 Concurrency

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

TODO: This is poorly written — fix

.

This is because the invocation of an operation not only offers up the operation's payload, but also the *continuation* of the calling computation. The handler for this operation is then free to do what it pleases with the continuation. For many effects, such as `getState`, nothing interesting happens to the continuation; in the case of `getState`, it is resumed with the value in state. But these continuations are first-class; they can resumed, sure, but also stored elsewhere or even thrown away. As such, by handling `Yield` operations, we easily pause and switch between several threads.

1.2.1 Simple Scheduling

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

```
interface Yield = yield : Unit

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

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

First note the simplicity of the `Yield` interface; we have one operation supported, which looks very boring; the operation `yield!` will just return unit — of course, it is

the way we *handle* yield that is more interesting.

```
-- Runs all of the LHS first, then the RHS.
scheduleA : {<Yield> Unit -> <Yield> Unit -> Unit}
scheduleA <yield -> m> <n> = scheduleA (m unit) n!
scheduleA <m> <yield -> n> = scheduleA m! (n unit)
scheduleA _ _ = unit

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

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

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

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

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

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

1.2.2 Forking New Processes

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

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

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

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

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

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

Notice the use of the `Queue` effect; we have to handle the computation `scheduleBF forker!` with a handler for `Queue` effects afterwards. Moreover, notice how concisely we can express the scheduler; this is due to the handler having access to te continuation of the caller, and treating it as a first-class object that can be stored elsewhere. We can see a diagram of how `scheduleBF` treats continuations in Figure 1.1, and a similar

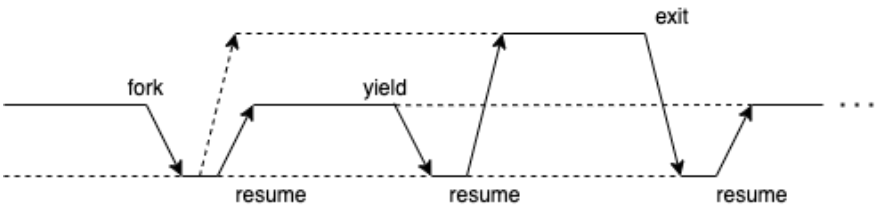

Figure 1.1: Breadth-First scheduling

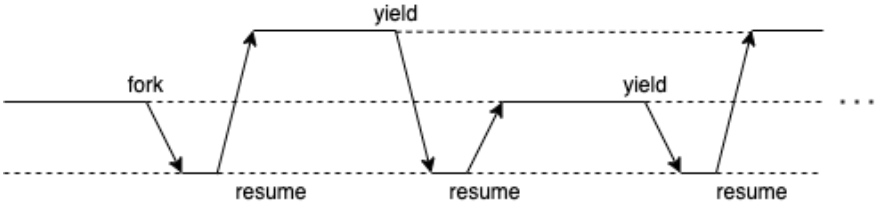

Figure 1.2: Depth-First scheduling

diagram of how the depth-first handling differs in Figure 1.2.

Chapter 2

Formalisation of Frank

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

Figure 2.1: Types

(constructors)	k
(commands)	c
(uses)	$m ::= x \mid f\ \overline{R} \mid m\ \overline{n} \mid \uparrow (n : A)$
(constructions)	$n ::= \downarrow m \mid k\ \overline{n} \mid c\ \overline{R}\ \overline{n} \mid \{e\}$
	$\mid$ **let** $f : P = n$ **in** $n' \mid$ **letrec** $\overline{f : P = e}$ **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 2.2: Terms

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

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

$$\textsc{T-App}\ \frac{\Sigma' = \Sigma \qquad (\Sigma \vdash \Delta_i \dashv \Sigma'_i)_i \qquad \Phi;\Gamma\,[\Sigma]\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to}\ [\Sigma']B\} \qquad (\Phi;\Gamma\,[\Sigma'_i]\vdash n_i : A_i)_i}{\Phi;\Gamma\,[\Sigma]\vdash m\ \overline{n} \Rightarrow B} \qquad \textsc{T-Ascribe}\ \frac{\Phi;\Gamma\,[\Sigma]\vdash n : A}{\Phi;\Gamma\,[\Sigma]\vdash \uparrow(n : A) \Rightarrow A}$$

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

$$\textsc{T-Switch}\ \frac{\Phi;\Gamma\,[\Sigma]\vdash m \Rightarrow A \qquad A = B}{\Phi;\Gamma\,[\Sigma]\vdash \downarrow m : B} \qquad \textsc{T-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad (\Phi;\Gamma\,[\Sigma]\vdash n_j : A_j)_j}{\Phi;\Gamma\,[\Sigma]\vdash k\ \overline{n} : D\ \overline{R}}$$

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

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

$$\textsc{T-LetRec}\ \frac{(P_i = \forall \overline{Z}_i.\{C_i\})_i \qquad (\Phi, \overline{Z}_i;\Gamma, \overline{f : P} \vdash e_i : C)_i \qquad \Phi;\Gamma, \overline{f : P}\,[\Sigma]\vdash n : B}{\Phi;\Gamma\,[\Sigma]\vdash \textbf{letrec}\ \overline{f : P = e}\ \textbf{in}\ n : B} \qquad \textsc{T-Adapt}\ \frac{\Sigma \vdash \Theta \dashv \Sigma' \qquad \Phi;\Gamma\,[\Sigma']\vdash n : A}{\Phi;\Gamma\,[\Sigma]\vdash \langle\Theta\rangle\ n : A}$$

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

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

Figure 2.3: Term Typing Rules

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

Figure 2.4: Runtime Syntax

$\boxed{\Phi; \Gamma\ [\Sigma] \vdash m \Rightarrow A}$ $\boxed{\Phi; \Gamma\ [\Sigma] \vdash n : A}$

$$
\text{T-FREEZE-USE}\quad \frac{\neg(\mathcal{E}\ \mathsf{handles}\ c) \qquad \Phi; \Gamma\ [\Sigma] \vdash \mathcal{E}[c\ \overline{R}\ \overline{w}] \Rightarrow A}{\Phi; \Gamma\ [\Sigma] \vdash \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil \Rightarrow A}
$$

$$
\text{T-FREEZE-CONS}\quad \frac{\neg(\mathcal{E}\ \mathsf{handles}\ c) \qquad \Phi; \Gamma\ [\Sigma] \vdash \mathcal{E}[c\ \overline{R}\ \overline{w}] : A}{\Phi; \Gamma\ [\Sigma] \vdash \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil : A}
$$

Figure 2.5: Frozen Commands

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

R-HANDLE

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

R-ASCRIBE-USE

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

R-ASCRIBE-CONS

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

R-LET

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

R-LETREC

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

R-ADAPT

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

R-FREEZE-COMM

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

R-FREEZE-FRAME-USE

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

R-FREEZE-FRAME-CONS

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

R-LIFT-UU

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

R-LIFT-UC

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

R-LIFT-CU

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

R-LIFT-CC

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

Figure 2.6: Operational Semantics

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

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

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

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

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

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

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

Figure 2.7: Pattern Binding

Chapter 3

Arbitrary Thread Interruption

3.1 Motivation

One important part of our asynchronous effect handling system is the ability to interrupt arbitrary computations. This is essential for pre-emptive concurrency, which relies on being able to suspend a computation *without* the computation yielding.

For instance, consider the two programs below;

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

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

We ideally want a multihandler that can run these two programs in parallel, such that the result will be `1 stop 2 stop 3 stop`; that is to say, the `stop` and `go` operations from `controller` can control the execution of `runner`.

3.2 Interruption with Yields

One way we can get this behaviour is using the `Yield` interface. This offers a single operation, `yield : Unit`. With this, we can write a multihandler `suspend`;

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

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

TODO: Maybe make more concise?

TODO: change to not be letter l cos it looks like a 1

Running `suspend runner! controller! nothing` then prints out `1 stop 1 2 stop 2 3 stop 3` as desired. So far so good; this works as planned. However, observe that we had to change the code of the original runner program to `yield` every time it prints. We would rather not have this requirement; the threads should be suspendable without knowing in advance they will be suspended, and thus without needing to explicitly `yield`. Furthermore, see that the `yield` operation adds no more information; it is just used as a placeholder operation; any operation would work. As such, we keep searching for a better solution.

3.3 Relaxing Catches

The key to this lies in the catchall pattern, $\langle x \rangle$, and the pattern binding rules of Figure 2.7.

Recall that the catchall pattern $\langle x \rangle$ matches either a value — e.g. `unit` — or a command that is handled in the surrounding ability. For instance, the pattern `<k>` in the code above matches either `unit` or `<yield -> k>`. The variable `k` is then bound to whatever this match is, leaving the (potentially) invoked effect unhandled. This is expressed in the B-CATCHALL-REQUEST rule in Figure 2.7.

Important to note is that only effects that are handled in that position are able to be caught. `runner` also makes use of the `print` effect, but these are not able to be caught by the catchall command. Formally, this is due to the fourth requirement of B-CATCHALL-REQUEST; that the command c that is invoked is a member of Ξ.

As such, we propose to remove this constraint from B-CATCHALL-REQUEST. The

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

Figure 3.1: Updated B-CatchAll-Request

resulting rule can then be seen in Figure 3.1. This lets us update the previous `suspend` code to the following, which yields the same results as last time;

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

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

TODO: Verify that this particular example really works.

Why does this work?

TODO: explain

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

3.4 Interrupting Arbitrary Terms

The approach of Section 3.3 can only interrupt command invocations. If `runner` were instead a sequence of pure computations[1], we would be unable to interrupt it; it does not invoke commands.

As such, we need to further change the pattern binding rules of Figure 2.7. This is to let us interrupt arbitrary computation terms. In Figure 3.2, we see an updated version of the runtime syntax; this allows for the suspension of arbitrary *uses*, being function applications and constructions.

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

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

Figure 3.2: Runtime Syntax, Updated with Suspension of Uses

$$
\text{B-CatchAll-Interrupt}\ \frac{\Sigma \vdash \Delta \dashv \Sigma'}{\langle x \rangle : \langle \Delta \rangle A \leftarrow !(m) \dashv [\Sigma]\ [\uparrow(\{m\} : \{[\Sigma']A\})/x]}
$$

Figure 3.3: Catching Interrupts rule.

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

TODO: Do we need to add interrupts to evaluation frames?

With this in mind, we now give the updated rule for the catchall pattern matching on interrupted terms (denoted $!(m)$). This can be seen in Figure 3.3. It expresses that an arbitrary suspended *use* can be matched against the computation pattern $\langle x \rangle$. The suspended computation $\{m\}$ is then bound to x, in a similar way to other B-CatchAll rules.

Figure 3.4 shows how uses m become interrupted. This rule supplements the operational semantics of Figure 2.6. It states that at any point, a use term can reduce to the same term but suspended. At this point, the term cannot reduce any further; observe that $!()$ is not an evaluation context. The term then blocks until it is handled away. Note how similar this is to regular command invocations, which block until their continuation is invoked.

TODO: Check that just uses is enough

TODO: Talk about how this achieves our goal.

The addition of this rule introduces non-determinism into the language; at any

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

Figure 3.4: Use interruption rule

point, a use can either step as normal (e.g. through the R-HANDLE rule), or it can be interrupted. An interrupted term $!(m)$ can no longer reduce; it is blocked until it is resumed by the R-HANDLE rule.

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

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

3.5 Interrupting In Practice

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

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

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

TODO: Rewrite above paragraph to be less camp

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

These `yield` effects are only inserted in a computation when the `Yield` interface is present in the ability of this computation. This is important, as it lets the programmer get fine-grained control over which computations can be interrupted and which cannot. Consider the example in Section 3.1; we want the `runner` computation to be controller by the `controller`. As such, we want the `runner` to be interruptible, whilst

the `controller` is not. We can reflect this by adding the `Yield` interface to the ability of the former computation.

Thus our example from before becomes;

TODO: updated example

Appendix A

Remaining Formalisms

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

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

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

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

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

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

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

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

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

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

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

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

Figure A.1: 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.2: Well-Formedness Rules

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

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

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

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

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

Figure A.3: Pattern Matching Typing Rules