Encapsulating Effects in Frank

Lukas Convent[1], Sam Lindley[2], Conor McBride[3], and Craig McLaughlin[2]

[1] University of Lübeck, Lübeck, Germany
`convent@isp.uni-luebeck.de`
[2] The University of Edinburgh, Edinburgh, UK
{`sam.lindley`,`craig.mclaughlin`}`@ed.ac.uk`
[3] University of Strathclyde, Glasgow, UK
`conor.mcbride@strath.ac.uk`

Abstract. Plotkin and Pretnar's effect handlers offer a versatile abstraction for modular programming with user-defined effects. Alas, many implementations are not as modular as they may at first seem. Naive composition of a pair of effect handlers, one a producer, and the other a consumer of an intermediate effect, leads to *effect pollution*: the intermediate effect leaks and external instances are accidentally captured.
We extend the Frank programming language with *adaptors*, which provide general remapping, and in particular hiding, of effect names in order to support effect encapsulation. Frank is a strict bidirectionally typed effect handler oriented programming language with a parsimonious effect type system in which effect polymorphism is almost always invisible.
As a case study, we compose a concurrent actor handler from a collection of more primitive effect handlers using Frank. The naive implementation without adaptors yields an actor handler that suffers from effect pollution. Using adaptors we define an unpolluted actor handler whose type is also five times shorter than that of the naive version.
As well as formalising adaptors, we also extend the formalism with other features necessary for the case study including polymorphic commands and a built-in top-level reference effect. We give a type system and an operational semantics and prove type soundness.

1 Introduction

Since Moggi's seminal work on monads [17] and Wadler's remarkably successful initiative to apply them in practice in the Haskell programming language [24], it has been apparent that programming abstractions for interpreting user-defined effects have much to offer. Algebraic effects were introduced by Plotkin and Power [20,21,22] as a refinement of monadic effects. A monadic effect is specified as a *concrete* implementation. An algebraic effect, on the other hand, is specified as an *abstract* interface of effectful operations along with a collection of laws that these operations should satisfy. Thus, algebraic effects support the fundamental encapsulation principle: program to an *interface*, not to an *implementation* [12].

Effect handlers were introduced by Plotkin and Pretnar [23] as a means for defining the implementation of a computation that makes use of algebraic effects.

They offer a versatile abstraction for modular programming with user-defined effects. Alas, many implementations are not as modular as they may at first seem. Naive composition of a pair of effect handlers, one a producer, and the other a consumer of an intermediate effect, leads to *effect pollution*: the intermediate effect leaks and external instances are accidentally captured.

In this paper we extend the Frank programming language with *adaptors*, which provide general remapping, and in particular hiding, of effect names in order to support effect encapsulation. Frank is a strict bidirectionally typed effect handler oriented programming language with a parsimonious effect type system in which effect polymorphism is almost always invisible.

Remark. We choose to emphasise effect handlers and de-emphasise algebraic effects because our effect handlers (in common with most implementations) do not enforce algebraicity. Standard deep handlers are guaranteed to implement algebraic effects if the effect interfaces are first-order, but many useful instances are higher-order [19]. Our handlers are shallow [12,10], so even first-order effect interfaces need not be given an algebraic interpretation.

We now outline a small example in order to illustrate the problem of composing two effect handlers. First, let us define two effect interfaces.

```
interface Reader S = ask      : S
interface Abort    = abort X : X
```

The `Reader S` effect is parameterised by a state type `S` and supports a single command `ask` that offers to return a value of type `S`. The `Abort` effect supports an infinite family of `abort` commands, one for every type X, each offering to return a value of type X (`abort` never returns a value so can take any type).

An effect handler interprets computations over the commands supported by its effect interfaces. Let us consider two handlers:

- a `reads` handler parameterised by a list of state values, which interprets values as themselves and `ask` by returning the next value from the list if it exists or by invoking `abort` if the list is empty; and
- a `maybe` handler that yields results in an option type, and which interprets a value u as `just` u and an `abort` command as `nothing`.

The `reads` handler transforms a `Reader S` computation into an `Abort` computation. The `maybe` handler transforms an `Abort` computation into a pure computation. It is natural to expect to be able to precompose `maybe` (`[Abort]` $\Rightarrow$ `[ ]`) with `reads` (`[Reader S]` $\Rightarrow$ `[Abort]`) to obtain an interpretation of the `Reader S` effect as a pure computation.

However, much of the power of effect handlers as a programming abstraction arises from the ability to forward any additional effects that are not explicitly handled. For instance, we may apply the composed transformation to a computation that has the `Abort` effect and so may perform `abort` commands. Although `reads` does not handle `abort` itself, it will forward `abort` to the surrounding context. Effects are therefore transformed as follows.

```
reads    [Abort, Reader S] ⇒ [Abort]
maybe    [Abort]           ⇒ [ ]
```

The composition `maybe` ∘ `reads` yields an effect transformation `[Abort, Reader S]` ⇒ `[ ]` (rather than `[Abort, Reader S]` ⇒ `[Abort]`) which consumes the external `Abort` effect. The computation passed to `maybe` ∘ `reads` may perform an `abort` command that is intended for an outer handler but is intercepted by `maybe`. The `Abort` effect has leaked out and polluted the effects of the composition.

To prevent such effect pollution we need a way of *hiding* intermediate effects so that external effects with the same name are not accidentally handled. There are a multiple ways to achieve this, which we discuss in more detail in Sect. 6 and Sect. 7. The approach we adopt in this paper is a generalisation and adaptation of a construct variously called *inject* [13,15] and *lift* [3] in the literature, which in this paper we call *mask*. The mask construct allows an intermediate effect to be hidden by explicitly shadowing it. Our generalisation, *adaptors*, support arbitrary remapping of effect names and may appear in types or in terms.

The paper makes the following contributions:

- We identify the effect pollution problem and how it manifests in various different approaches to implementing effect handlers (Sect. 1 and Sect. 3).
- We implement [7] a range of extensions to Frank including polymorphic commands (Sect. 2), ML-style references (Sect. 2) and adaptors (Sect. 3).
- We present adaptors as a solution to the effect pollution problem and hence a means for obtaining effect encapsulation (Sect. 3).
- As a case study, we compose a concurrent actor handler from a collection of more primitive effect handlers using Frank. The naive implementation without adaptors yields an actor handler that suffers from effect pollution. Using adaptors we define an unpolluted actor handler whose type is also five times shorter than that of the naive version (Sect. 4).
- We formalise the semantics of Frank extended with adaptors and polymorphic commands. We present the first direct semantics for Frank (previous work gave a semantics via translation to a simpler language [16]) and prove type soundness (Sect. 5).

Section 6 discusses variations and extensions of adaptors. Section 7 discusses related work. Section 8 concludes.

The effect pollution problem was originally identified by the first author in his master's dissertation [6] when attempting to compose handlers in order to implement rich concurrency abstractions. Many of the ingredients of this paper (polymorphic commands, direct semantics, and ML-style references) were also introduced there. This paper additionally provides a solution to the effect pollution problem and a case study illustrating the power of effect encapsulation.

2 The Frank Programming Language

Frank [16] is a strict bidirectionally typed effect handler oriented programming language. If we disregard effects, then Frank looks quite like a standard typed functional programming language such as ML or Haskell.

Data. Primitive data types include integers (`Int`) and characters (`Char`). Algebraic data types are defined in the standard way.

```
data Zero =
data Unit = unit
data Bool = false | true
data List X = nil | cons X (List X)
data Maybe X = nothing | just X
data Pair X Y = pair x y
```

Syntactic sugar for lists is built in. For instance:

```
[] ≡ nil    x::xs ≡ cons xs xs    [1,2] ≡ 1::2::[]
```

The `String` type is an alias for `List Char`.

Operators. In place of plain functions, Frank provides *operators*, which generalise both functions and effect handlers. Consider the `map` function in Frank:

```
map : {{X -> Y} -> List X -> List Y}
map f []        = []
map f (x :: xs) = f x :: map f xs
```

Other than the type signature, the definition looks like the equivalent definition in ML defined by pattern matching. Moreover, we can apply it in the usual way

```
map {n -> n+1} [1, 2, 3] ⟹ [2, 3, 4]
```

where $\Longrightarrow$ denotes evaluation. However, operators in Frank are n-ary, not curried (despite the syntax). Thus, `map` is a binary function that takes a unary function as its first argument and a list as its second argument.

More interestingly, `map` is implicitly effect-polymorphic. The type

```
{{X -> Y} -> List X -> List Y}
```

is really syntactic sugar for:

```
{{X -> [ε| ]Y} -> List X -> [ε| ]List Y}
```

where ε is an effect variable which is fresh for this particular type signature. The effects performed by `map` are the same as those performed by its first argument. Frank implicitly inserts ε on every function return type.

The general form of an operator type is as follows

`{<`Δ_1`>`A_1 `->` ... `->` `<`Δ_k`>`A_k `->` `[`Σ`]`B`}`

having k argument types `<`Δ_1`>`A_1, ..., `<`Δ_k`>`A_k and one return type `[`Σ`]`B. If we ignore the Δ_is and Σ, then we can view an operator type as a pure k-ary function type. The value types A_1 to A_k are the types of the arguments, and the value type B is the return type of the function.

The *ability* Σ describes the effects that the operator may perform. It includes at most one (possibly implicit) effect variable and a collection of effect instances. As operators generalise effect handlers (as well as functions), the arguments to an operator are themselves computations rather than values. The *adjustments*

$\Delta_1, ..., \Delta_k$ describe which effects may be handled in each argument computation. All of the effects in the ambient ability of the result type may also occur in argument computations; these will be forwarded to an outer handler.

Recall the two effect interfaces we defined in the introduction.

```
interface Reader S = ask     : S
interface Abort    = abort X : X
```

The `maybe` handler is defined as follows.

```
maybe : {<Abort>X -> Maybe X}
maybe <abort -> _> = nothing
maybe x            = just x
```

It takes a computation which may abort and returns an optional value. The command pattern `<abort -> _>` matches any instance of the `abort` command invoked by the argument computation. The wildcard pattern `_` binds the continuation at the point in which `abort` is invoked, but as this handler is implementing aborting, the continuation is ignored. The variable pattern `x` matches a final return value. We can also define other handlers for the `Abort` effect.

```
catch : {<Abort>X -> {X} -> X}
catch x            _ = x
catch <abort -> _> h = h!
```

The `catch` handler takes a suspended computation `h` as a second argument. The `abort` command is interpreted by invoking `h` (`!` denotes nullary application). Here are some examples of applying `maybe` and `catch`.

Type	**Program**	**Result**
`Maybe Int`	`maybe 42`	`just 42`
`Maybe Int`	`maybe (let x = abort! in 42)`	`nothing`
`Int`	`catch 42 {0}`	`42`
`Int`	`catch (let x = abort! in 42) {0}`	`0`

The `reads` handler is defined as follows.

```
reads : {List S -> <Reader S>X -> [Abort]X}
reads []        <ask -> k> = abort!
reads (s :: ss) <ask -> k> = reads ss (k s)
reads _         x          = x
```

It takes a list and a reader computation and it may abort. The command pattern `<ask -> k>` binds `k` to the continuation. Here are some example of applying `reads` and combining it with `maybe`.

Type	**Program**	**Result**
`[Abort]X`	`reads [1,2] (ask! + ask!)`	`3`
`[Abort]X`	`reads [1,2] (ask! + ask! + ask!)`	`abort!`
`Maybe X`	`maybe (reads [1,2] (ask! + ask! + ask!))`	`nothing`
`Maybe X`	`maybe (reads [1,2] abort!)`	`nothing`

The final program suffers from effect pollution. The goal is for `Abort` to be an intermediate effect that is encapsulated; not one that is available for the argument computation to perform. We return to this example in the next section.

Operators as Functions, Handlers, and Multihandlers. We say that an operator is a *function* if none of its argument types includes a non-trivial adjustment. Conversely we say that it is an *effect handler* if at least one of its argument types includes a non-trivial adjustment. We say that an operator is a *multihandler* [16] if more than one of its argument types includes a non-trivial adjustment. (In this paper we do not consider multihandlers, but they are a useful feature of Frank.)

The Case Operator. In Frank pattern matching is built-in to operation definitions, but there is also a case operator, which is defined as reverse application.

```
case : {X -> {X -> Y} -> Y}
case x f = f x
```

For instance, we can rewrite the `map` function as follows.

```
map' : {{X -> Y} -> List X -> List Y}
map' f xs = case xs { []         -> []
                    | (x :: xs) -> f x :: map' f xs }
```

The second argument to `case` here is an anonymous operator. Anonymous operators support the same patterns as named operator definitions. An anonymous operator definition is enclosed in braces and its (zero or more) clauses are separated by the vertical bar symbol.

Implicit Effect Polymorphism. As with `map`, each of our other operators exhibit some implicit effect polymorphism. Here are the full type signatures for each of the handlers we have seen so far.

```
maybe : {<Abort>X -> [ε| ]Maybe X}
catch : {<Abort>X -> {[ε| ]X} -> [ε| ]X}
reads : {List S -> <Reader S>X -> [ε|Abort]X}
```

In each type signature, the implicit polymorphic type variable ε is attached to each return type. We can explicitly name effect variables. For instance, we could have given `catch` the following α-equivalent type signature.

```
{<Abort>X -> {[E| ]X} -> [E| ]X}
```

Effect Shadowing. An ability may contain multiple instances of the same effect interface. In this case, the rightmost one is the one that is active. The others come into play once the rightmost one has been handled. Effect shadowing allows us to instantiate ε in the signature of `catch` in order to re-raise an exception [13].

```
maybe (catch abort! {print "oops"; abort!}) ⟹ nothing
```

Specifically, ε is instantiated to include an `Abort` instance in order to match the argument type of `maybe` and thus the argument `abort!` passed to `catch` must be typechecked against an ability containing two instances of `Abort`.

The order of instances of the same effect interface is important. The order of instances of different effect interfaces relative to one another is not. Thus, an ability denotes a finite map from interface names to lists of instances (possibly extended through an effect variable).

Monomorphic Effects. We can give operators monomorphic effects. For instance, we can define a restricted version of `catch` whose handler argument is pure.

```
pureCatch : {<Abort>X -> {[0| ]X} -> [0| ]X}
pureCatch x              _ = x
pureCatch <abort -> _> h = h!
```

Here `0` indicates that the ambient ability and hence also the exception handler argument is pure. This prevents `pureCatch` from being able to re-raise an exception, for instance.

Polymorphic Commands. The original version of Frank [16] supports only monomorphic commands. We have now extended it with polymorphic commands [12] like `abort` which can be invoked at any type. Previously we had to do some gymnastics to get the same effect.

```
interface Abort' = aborting : Zero

abort' : {[Abort]X}
abort'! = case aborting! {}
```

Polymorphic commands are necessary for more interesting examples like ML-style dynamically allocated references.

ML-style References. The Frank implementation now supports a special top-level effect for ML-style dynamically allocated references.

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

Both the data type `Ref X` and the interface `RefState` are built-in, as Frank is not expressive enough to define them internally.

Console. Similarly to `RefState`, Frank supports another special built-in effect for console I/O.

```
interface Console = inch : Char
                  | ouch : Char -> Unit
```

The `print` operator takes a string and outputs it to the console.

```
print : {String -> [Console]Unit}
print s = map ouch s; unit
```

The `RefState` and `Console` effects in Frank have a status similar to the IO monad in Haskell.

Program Entry Point. The default entry-point for a Frank program is the `main` operator, which may perform top-level effects and return a value.

```
main : {[0|Console]Unit}
main! = print "do be do be do"
```

The above program prints out the string "`do be do be do`" to the console and returns the unit value. The ability of its return type ensures that it may use the top-level `Console` effect but no others.

3 Adapting Effects

In this section we discuss the effect pollution problem and how to resolve it, obtaining effect encapsulation via adaptors.

Mask. Let us return to our running example of precomposing `maybe` with `reads`. We can do so as follows.

```
bad : {List S -> <Reader S, Abort>X -> Maybe X}
bad ss <m> = maybe (reads ss m!)
```

Both `Reader S` and `Abort` are handled. The catch-all pattern `<m>` binds the second argument to a suspended computation `m` of type `{[Reader S, Abort]X}`. The `ask` command is handled by yielding `just v` for some value `v` if the input is non-empty

```
bad [1,2] (ask! + ask!) ⟹ just 3
```

and `nothing` if the input is exhausted:

```
bad [1]   (ask! + ask!) ⟹ nothing
```

Alas, as indicated by its type, `bad` also exhibits additional behaviour. As well as handling any `abort` command raised by the `reads` handler, it also handles uses of `abort` within the argument computation.

```
bad [1,2] (ask! + abort!) ⟹ nothing
```

This breaks abstraction as an invocation of `abort` never reaches the outer context (in order to be handled there) but is instead intercepted by `bad`. Using an *adaptor* prohibits the unintentional interception of such external effects.

```
good : {List S -> <Reader S>X -> Maybe X}
good ss <m> = maybe (reads ss (<Abort> m!))
```

The term `<Abort> m!` is executed under ambient `[`ε`|Abort,Reader S]`. The adaptor `<Abort>` *adapts* the ambient in which `m!` is executed by hiding the `Abort` effect, resulting in the ambient `[`ε`|Reader S]`. This ambient matches exactly the ability of `m`. Consequently, adaptors have computational content. Hiding an effect instance skips the nearest dynamically enclosing handler for the effect. The `Abort` adaptor ensures that the `maybe` handler cannot capture `abort` commands raised by the argument computation. So

```
good [1,2] (ask! + abort!) ⟹ abort!
```

whereas:

```
bad  [1,2] (ask! + abort!) ⟹ nothing
```

The plain `Abort` adaptor above is an instance of mask, which transforms the ambient ability to obtain the ability for `m!` by masking out the rightmost instance of `Abort` providing a means for hiding effects reminiscent of de Bruijn representations for bound names.

Remark. Computing the local ability by *restricting* the ambient ability is natural for a bidirectionally typed language such as Frank, where effect types always flow inwards. An alternative but equivalent view that is arguably more natural for a language with Hindley-Milner type inference, where effect types flow outwards, is of *extending* the ability of `m!` to include a fresh instance of `Abort`. In this sense one is injecting [13] or lifting [3] the local ability into the ambient ability.

Adaptors as Finite Maps. Frank's adaptors generalise mask to an arbitrary finite map from the effects of the ambient ability to the effects of the supplied computation. Effect instances can be masked (as in `good`), re-ordered, and duplicated.

In general an adaptor is given by a sequence of adaptor components of the form I`(`S `->` S'`)` such that each I must be distinct and S ranges over patterns that bind the instances of I. Examples:

`I(s x -> s)`	erase the first instance of `I` (mask)
`I(s x y -> s y)`	erase the second instance of `I`
`I(s x y -> s y x)`	swap the first two instances of `I`
`I(s x -> s x x)`	duplicate the first instance of `I`
`I(s -> s)`	identity at `I`

For the common case of mask we omit the map entirely: `I` is syntactic sugar for `I(s x -> s)`. Technically, mask in combination with handlers is sufficient to express general adaptors, but only through a global transformation. For instance, Biernacki et al. [3] express swapping of effects using mask (which they call “lift”).

Adaptor Adjustments. A distinctive feature of Frank is that the effects associated with an operator argument describe an adjustment to the ambient ability. This has two clear benefits over a more conventional type system. First, it is often more parsimonious, as the programmer need only specify the change to the ambient ability rather than the entire ability. Second, it explicitly distinguishes those effects that are handled from those that are forwarded. An unfortunate side-effect of this scheme is that if an interface appears both in the return type and in the argument type then the actual type of an argument computation will include two instances of the interface, which is usually not desirable.

Let us suppose we wish to define an operator that acts as a `Reader Int` transformer adding one to each integer that is read. We might write the following

```
inc' : {<Reader Int>X -> [Reader Int]X}
inc' <ask -> k> = let n = ask! in inc' (k (n + 1))
inc' x          = x
```

which may be combined with another `Reader` handler without ado:

```
reads [1,2] (inc' (ask! + ask!)) ⟹ 5
```

However, suppose we run `inc'` twice.

```
incinc' : {<Reader Int, Reader Int>X -> [Reader Int]X}
incinc' <m> = inc' (inc' m!)
```

For this function to type check we have had to insert an extra copy of `Reader Int` in the argument adjustment. Using an adaptor we can obtain a type with just one instance of `Reader` in the adjustment.

```
incinc'' : {<Reader Int>X -> [Reader Int]X}
incinc'' <m> = inc' (inc' (<Reader(s x y -> s y)> m!))
```

However, the root cause of the problem is in `inc'` which still accepts an argument with two copies of `Reader Int`. Adaptor adjustments are a means for wiring an adaptor into an operator and its type. Consider the following variation.

```
inc : {<Reader|Reader Int>X -> [Reader Int]X}
inc <ask -> k> = let n = ask! in inc (k (n + 1))
inc x          = x
```

The difference between `inc` and `inc'` is that the former applies the `Reader` adaptor to the ambient ability before applying the rest of the extension `Reader Int`. As a consequence, the type of an argument includes only one copy of `Reader Int` and we can compose `inc` with itself without need for any further inline adapters.

```
incinc : {<Reader|Reader Int>X -> [Reader Int]X}
incinc <m> = inc' (inc' m!)
```

The general form for an adjustment is Θ`|`Ξ where Θ is an adaptor and Ξ is an *extension* (the part of an adjustment we have seen up to now, which is added to the ambient ability). The action of an adjustment on an ability is to first apply the adaptor and to then apply the extension. By default we assume the adaptor is the identity and write `<`Ξ`>` as syntactic sugar for `< |`Ξ`>`.

When generalising `incinc` to an operator that composes an arbitrary number of `inc` operators, the problem of effect pollution becomes even more apparent. Without adaptors, it would require an unbounded number of effect instances. With adaptors, we can define such a composed operator as follows.

```
incN : {Int -> <IChoice|IChoice>X -> [IChoice]X}
incN 0 <m> = m!
incN n <m> = inc (incN (n-1) m!)
```

4 Concurrency in Frank

In this section we implement concurrent actors with mailboxes in Frank. We do so in stages, by composing together several handlers.

Effect Interfaces. Let us begin with the high-level actor effect interface `Actor M` for an actor computation whose mailbox stores messages of type `M`.

```
interface Actor M = spawn N : {[Actor N]Unit} -> Pid N
                  | self    : Pid M
                  | send N  : N -> Pid N -> Unit
                  | recv    : M
```

The `Pid M` data type represents a process ID for a process with a mailbox of type `M`. We define it later in this section. The `Actor M` interface supports four commands: `spawn` runs the supplied thunk as a new process for which a fresh process ID is generated and returned, `self` returns the process ID of the current process, `send` places a message in the mailbox of the specified process, and `recv` removes and returns a message from the mailbox of the current process.

As a first step we define a cooperative concurrency effect interface `Co`.

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

It supports two commands: `fork` runs the supplied thunk as a new process and `yield` allows control to switch to another process. A suspended computation passed to `fork` may itself fork and yield, hence the interface is recursive.

In order to implement cooperative concurrency we store suspended processes in a queue via the effect interface `Queue S` for a queue with elements of type `S`.

```
interface Queue S = enqueue : S -> Unit
                  | dequeue : Maybe S
```

It supports two commands: `enqueue` pushes a value onto the queue and `dequeue` pops a value `x` from the queue and returns `just x` if the queue is non-empty and returns `nothing` if the queue is empty.

Now we provide implementations for each of our three interfaces in turn from low-level to high-level.

Queues. One might imagine various different implementations for queues; here we focus on one based on a zipper [11] structure.

```
data ZipQ S = zipq (List S) (List S)
```

A queue is represented as a pair of lists: the first is the front of the queue, supporting amortised constant time popping from the front of the queue; the second is the back of the queue in reverse, supporting constant time pushing to the back of the queue.

An empty queue is a pair of empty lists.

```
emptyZipQ : {ZipQ S}
emptyZipQ! = zipq [] []
```

In order to implement the queue commands we define a handler.

```
runFifo : {ZipQ S -> <Queue S>X -> Pair X (ZipQ S)}
runFifo (zipq front back)        <enqueue x -> k> =
  runFifo (zipq front (x :: back)) (k unit)
runFifo (zipq [] [])             <dequeue -> k> =
  runFifo emptyZipQ! (k nothing)
runFifo (zipq [] back)           <dequeue -> k> =
  runFifo (zipq (rev back) []) (k dequeue!)
runFifo (zipq (x :: front) back) <dequeue -> k> =
  runFifo (zipq front back) (k (just x))
runFifo queue                    x =
  pair x queue
```

The `runFifo` handler takes an initial queue as a parameter before interpreting a `Queue` computation as a FIFO queue, returning a pair of the final return value and the final contents of the queue.

Schedulers. Using our queue interface we can define schedulers for concurrent processes. The processes we store in the queue will themselves need to be able to manipulate the queue, so we represent them through a recursive data type.

```
data Proc = proc {[Queue Proc]Unit}
```

We now define two helper functions for enqueueing a process and waking the process at the head of the queue if it exists.

```
enqProc : {[Queue Proc]Unit} -> [Queue Proc]Unit
enqProc p = enqueue (proc p)

wakeProc : {[Queue Proc]Unit}
wakeProc! = case dequeue! { (just (proc x)) -> x!
                          | nothing         -> unit }
```

Two canonical schedulers are breadth-first (forked processes are deferred) and depth-first (forked processes are run eagerly). We use the former for our actor implementation, but we could easily swap in an alternative if desired.

```
runBF : {<Co>Unit -> [Queue Proc]Unit}
runBF <yield -> k>  = enqProc {runBF (k unit)};
                      wakeProc!
runBF <fork p -> k> = enqProc {runBF (<Queue> p!)};
                      runBF (k unit)
runBF unit          = wakeProc!
```

An adaptor is required in order to allow a forked process to have the right type. Without the adapter, the argument type to the scheduler would be polluted and have to become `<Co [Queue Proc]>`. Allowing processes to do their own low-level manipulation of the process queue breaks abstraction.

Actors. We now define actors on top of cooperative concurrency, queues, and references. Let us first define a data type for process IDs.

```
data Pid X = pid (Ref (ZipQ X))
```

A process ID stores its associated mailbox as a mutable reference to a queue.

The core of the actor implementation is given by the `runActor` handler.

```
runActor : {Pid X -> <Actor X>Unit ->
             [Co [RefState], RefState]Unit}
runActor mine    <self -> k> = runActor mine (k mine)
runActor mine    <spawn you -> k> =
  let yours = pid (new (emptyZipQ!)) in
  fork {runActor yours (<RefState, Co> you!)};
  runActor mine (k yours)
runActor (pid m) <recv -> k> =
  case (runFifo (read m) dequeue!)
    { (pair nothing _)  -> yield!;
                            runActor (pid m) (k recv!)
    | (pair (just x) q) -> write m q;
                            runActor (pid m) (k x) }
runActor mine    <send (pid m) x -> k> =
  case (runFifo (read m) (enqueue x))
        { (pair _ q) -> write m q;
                         runActor mine (k unit) }
runActor mine    unit = unit
```

It interprets an actor communication using the concurrency interface and references. An adaptor masks the `RefState` and `Co` effects for the spawned process.

Now we compose all of our handlers together to obtain a handler that implements computations.

```
act : {<Actor X>Unit -> [RefState]Unit}
act <m> = case (runFifo emptyZipQ!
               (runBF (<Queue>
               (runActor (pid (new (emptyZipQ!)))
                         (<Co> m!)))))
               { (pair x _) -> x }
```

We use adaptors to make the argument computation compatible with the concurrency interface and the output of `runActor` compatible with the queue interface.

To test the actor implementation we implement a classic example that spawns a chain of processes and passes a message along the chain.

```
spawnMany : {Pid String -> Int ->
               [Actor String [Console], Console]Unit}
spawnMany p 0 = send "do be do be do" p
spawnMany p n = spawnMany (spawn {let x = recv! in
                                  print ".";
                                  send x p}) (n-1)

chain : {[Actor String [Console], Console]Unit}
chain! = spawnMany self! 640; let msg = recv! in
         print "\n"; print msg; print "\n"
```

Now if we run

```
main : {[0|Console, RefState]Unit}
main! = act (<RefState> chain!)
```

each spawned process prints out a dot as the message is passed along and the message received by the top-level process (`"do be do be do"`) is printed out after having been transmitted all the way along the chain.

Actors without Adaptors. To illustrate how bad things get without adaptors, let us contrast the type signature we gave for `act`

```
{<Actor X>Unit -> [RefState]Unit}
```

with the one we obtain by removing all of the adaptors and adjusting type signatures of constituent handlers accordingly [6]:

```
{<Queue (Proc [RefState]),
  Co [Queue (Proc [RefState]), RefState],
  Actor X [RefState, Queue (Proc [RefState]),
           Co [Queue (Proc [RefState]),
               RefState]]>Unit -> [RefState]Unit}
```

The implementation details have leaked into the type signature. Worse, the dynamic behaviour is incorrect as the low-level handlers will handle `Queue` and `Co` effects in the argument computation.

State Actors. Because `act` uses `RefState` internally, we need to use an additional adaptor if we wish to run an actor that itself uses `RefState`. For instance, suppose we replace the counter in `spawnMany` with an integer reference cell and correspondingly change the signature of chain to

```
{[Actor String [Console, RefState], Console, RefState]Unit}
```

then we must adapt the call to `chain` as follows:

```
main : {[0|Console, RefState]Unit}
main! = <RefState(s x -> s x x)> (act (<RefState> chain'!))
```

The outer adapter ensures that the two instances of `RefState` are mapped to the same built-in `RefState` effect.

5 Frank Formalised

In this section we formalise the syntax, typing rules, and operational semantics for Frank including adaptor and polymorphic command extensions. (For simplicity we do not account for built-in effects such as `Console` and `RefState`.) The syntax and typing rules extend those of Lindley et al. [16]. Whereas they give an operational semantics via a translation into a core language, our operational semantics is direct (following the first author's master's dissertation [6]). We prove a type soundness property for the extended system.

Notation. We write either overlines ($\overline{M}$) or explicit indexing ($(M)_i$) for a list of zero or more copies of M indexed by i.

5.1 Syntax

Types. The types (Fig. 1) are divided into value types (A) and computation types (C). Value types are data types ($D\ \overline{R}$), suspended computation types ($\{C\}$), or value type variables (X). Computation types are built from zero or more argument types (T) and one return type (G). A computation type

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

(value types)	$A, B ::= D\ \overline{R}$	(seeds)	$\sigma ::= \emptyset \mid E$
	$\quad \mid\ \{C\} \mid X$	(abilities)	$\Sigma ::= \sigma \mid \Xi$
(computation types)	$C ::= \overline{T \to}\ G$	(extensions)	$\Xi ::= \cdot \mid \Xi, I\ \overline{R}$
(argument types)	$T ::= \langle \Delta \rangle A$	(adaptors)	$\Theta ::= \cdot \mid \Theta, I(S \to S')$
(return types)	$G ::= [\Sigma] A$	(adjustments)	$\Delta ::= \Theta \mid \Xi$
(type binders)	$Z ::= X \mid [E]$	(interface patterns)	$S ::= s \mid S\ x$
(type arguments)	$R ::= A \mid [\Sigma]$	(type environments)	$\Gamma ::= \cdot \mid \Gamma, x : A$
(polytypes)	$P ::= \forall \overline{Z}.A$		$\quad \mid \Gamma, f : P$

Fig. 1. Types

(uses)	$m ::= x \mid f\ \overline{R} \mid m\ \overline{n} \mid \uparrow(n : A)$
(constructions)	$n ::= \downarrow m \mid k\ \overline{n} \mid c\ \overline{R}\ \overline{n} \mid \{e\}$
	$\quad \mid\ \mathbf{let}\ f : P = n\ \mathbf{in}\ n' \mid \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n$
	$\quad \mid\ \langle \Theta \rangle\ n$
(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)	$p ::= k\ \overline{p} \mid x$

Fig. 2. Terms

has argument types $\langle \Theta_1 | \Xi_1 \rangle A_1, \ldots, \langle \Theta_k | \Xi_k \rangle A_k$ and return type $[\Sigma]B$. A computation of type C must handle effects in Ξ_i on its i-th argument; all arguments are handled simultaneously. As a result it returns a value of type B and may perform effects in ambient ability Σ. The i-th argument may perform effects in ambient ability Σ remapped by adaptor Θ_i and augmented by extension Ξ_i.

Effect Polymorphism with an Invisible Effect Variable. Consider the type of `map`:

$$\{\{X \to Y\} \to \mathit{List}\ X \to \mathit{List}\ Y\}$$

Modulo the braces around the function types, this is the same type a functional programmer might expect to write in a language without support for effect typing. In fact, this type desugars into:

$$\{\langle \cdot | \cdot \rangle \{\langle \cdot | \cdot \rangle X \to [\varepsilon | \cdot] Y\} \to \langle \cdot | \cdot \rangle (\mathit{List}\ X) \to [\varepsilon | \cdot](\mathit{List}\ Y)\}$$

We adopt the convention that the identity adaptor/extension $\cdot$ may be omitted from adaptors/extensions and argument types.

$$\langle \Xi \rangle A \equiv \langle \cdot | \Xi \rangle A \qquad\qquad A \equiv \langle \cdot | \cdot \rangle A$$

Similarly, we adopt the convention that effect variables may be omitted from abilities and return types.

$$I_1\ \overline{R_1}, \ldots, I_k\ \overline{R_k} \equiv \varepsilon | I_1\ \overline{R_1}, \ldots, I_k\ \overline{R_k} \qquad\qquad A \equiv [\varepsilon | \cdot] A$$

Here ε is a distinguished effect variable, the *implicit effect variable* that is fresh for every type signature in a program. This syntactic sugar ensures that we need never write the implicit effect variable ε in a Frank program. In adaptors we write I as syntactic sugar for the common case of mask, i.e., $I(s\ x \to s)$.

Type binders may be value binders (X) or effect binders ($[E]$); polytypes may be polymorphic in both. Though we avoid effect variables in source code, we are entirely explicit about them in the abstract syntax and type system.

Data Types and effect interfaces are defined globally. A definition for data type $D(\overline{Z})$ is a collection of data constructor signatures of the form $k : \overline{A}$, where $\overline{A}$ may depend on $\overline{Z}$. Each data constructor belongs to a single data type. A definition for effect interface $I(\overline{Z})$ consists of a collection of command signatures of the form $c : \forall \overline{Z'}.\overline{A} \to B$, denoting that c is polymorphic in type variables $\overline{Z'}$, takes arguments of types $\overline{A}$, and returns a value of type B. The types $\overline{A}$ and B may depend on $\overline{Z}$ and $\overline{Z'}$. Each command belongs to a single interface.

Type environments distinguish monomorphic and polymorphic variables.

Effect Parameters with an Invisible Effect Variable. In the case that the first parameter of a data type or effect interface definition is its only effect variable ε, we may omit it from the definition.

An ability is a collection of effect instances initiated either with the empty ability $\emptyset$ (yielding a *closed* ability) or an effect variable E (yielding an *open* ability). For each interface, the order of its instances in an ability is important, as duplicates are permitted, in which case the rightmost instance is the one that will be handled first. Closed abilities can be used to enforce purity. In source code we write $\emptyset$ as `0`. An adjustment modifies the ambient ability. The action of an adjustment $\Delta = \Theta|\Xi$ on an ability Σ is as follows.

$$(\Theta|\Xi)(\Sigma) = \Xi(\Theta(\Sigma))$$

First the adapter Θ is applied to Σ and then the extension Ξ is applied to the result. The action of an extension on an ability is quite direct: each effect instance is added to the ability.

$$\cdot(\Sigma) = \Sigma \qquad\qquad (\Xi, I\ \overline{R})(\Sigma) = (\Xi(\Sigma)), I\ \overline{R}$$

The action of an adaptor on an ability is pointwise.

$$\cdot(\Sigma) = \Sigma \qquad (\Theta, I(S \to S'))(\Sigma) = I(S \to S')(\Theta(\Sigma))$$

The adaptor component for each interface is used to transform the ability.

$$I(S \to S')(\Sigma) = (\mathsf{remap}\ S'\ (\mathsf{inst}\ S\ (\Sigma \mathbin{@} I)\ \emptyset))(\Sigma - I)$$

We express this transformation in terms of four auxiliary functions.

– $\Sigma - I$ returns Σ with all instances of I removed:

$$\sigma|\cdot - I = \sigma|\cdot \qquad \Sigma, I\ \overline{R} - I = \Sigma - I \qquad \Sigma, I'\ \overline{R} - I = \Sigma - I, I'\ \overline{R}, \quad \text{if } I \neq I'$$

– $\Sigma \mathbin{@} I$ returns the extension Ξ such that $\Xi(\Sigma - I) = \Sigma$:

$$\sigma| \cdot @I = \cdot \qquad \Sigma, I\ \overline{R} \mathbin{@} I = \Sigma \mathbin{@} I, I\ \overline{R} \qquad \Sigma, I'\ \overline{R} \mathbin{@} I = \Sigma \mathbin{@} I, \quad \text{if } I \neq I'$$

– $\mathsf{inst}\ S\ \Xi\ \rho$ pattern matches S against the interfaces in Ξ and extends environment ρ with the resulting bindings:

$$\begin{aligned} \mathsf{inst}(s, \Xi, \rho) &= \rho[s \mapsto \Xi] \\ \mathsf{inst}(S\ x, (\Xi, I\ \overline{R}), \rho) &= \mathsf{inst}(S, \Xi, \rho[x \mapsto I\ \overline{R}]) \end{aligned}$$

– $\mathsf{remap}\ S\ \rho$ yields the extension obtained by instantiating S with ρ:

$$\begin{aligned} \mathsf{remap}(s, \rho) &= \rho(s) \\ \mathsf{remap}(S\ x, \rho) &= \mathsf{remap}(S, \rho), \rho(x) \end{aligned}$$

The action of adaptors on abilities is partial. We rely on the typing rules to ensure that adaptor components $I(S \to S')$ are well-formed in that S' only mentions variables that are bound in S, and that the length of S is no longer than the length of the list of instances for I in the ambient ability.

Whereas, abilities may contain duplicate effect instances, both the adaptor and extension components of an adjustment must not.

Terms. Frank follows a bidirectional typing discipline [18]. Thus terms (Fig. 2) are subdivided into *uses* (ranged over by m) whose types are inferred, and *constructions* (ranged over by n) which are checked against a type. Uses comprise monomorphic variables (x), polymorphic variable instantiations ($f\ \overline{R}$), applications ($m\ \overline{n}$) and type ascriptions ($\uparrow(n : A)$). Constructions comprise uses ($\downarrow m$), data constructor instances ($k\ \overline{n}$), command invocations ($c\ \overline{R}\ \overline{n}$), suspended computations ($\{e\}$), polymorphic let ($\mathbf{let}\ f : P = n\ \mathbf{in}\ n'$), mutual recursion ($\mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n$), and adaptors ($\langle\Theta\rangle\ n$). For clarity in the formalism we explicitly mark the injections of a use into a construction ($\downarrow m$) and a construction into a use ($\uparrow(n : A)$); in actual Frank code we always omit $\downarrow$ and $\uparrow$. We write ! as syntactic sugar for an empty sequence of constructions.

A computation is defined by a sequence of pattern matching clauses ($\overline{\overline{r} \mapsto n}$). Each pattern matching clause takes a sequence of computation patterns ($\overline{r}$). A computation pattern is either a standard value pattern (p), a request pattern ($\langle c\ \overline{p} \to z\rangle$), which matches command c if its arguments match $\overline{p}$ and binds the continuation to z, or a catch-all pattern $\langle x\rangle$, which matches any value or handled command, binding it to x. A value pattern is either a data constructor pattern ($k\ \overline{p}$) or a variable pattern (x).

5.2 Typing Rules

The typing rules are given in Fig. 3. The inference judgement $\Gamma\ [\Sigma]\!\!\vdash m \Rightarrow A$ states that in type environment Γ with ambient ability Σ, we can infer that use m has type A. The checking judgement $\Gamma\ [\Sigma]\!\!\vdash n : A$ states that in type environment Γ with ambient ability Σ, construction n has type A. The auxiliary judgement

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

T-Var
$$\frac{x : A \in \Gamma}{\Gamma\,[\Sigma]\!\!\vdash x \Rightarrow A}$$

T-PolyVar
$$\frac{f : \forall \overline{Z}.A \in \Gamma}{\Gamma\,[\Sigma]\!\!\vdash f\ \overline{R} \Rightarrow A[\overline{R}/\overline{Z}]}$$

T-App
$$\frac{\begin{array}{c}\Sigma' = \Sigma \qquad \Delta_i(\Sigma') = \Sigma_i \\ \Gamma\,[\Sigma]\!\!\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to}\ [\Sigma']B\} \qquad (\Gamma\,[\Sigma_i]\!\!\vdash n_i : A_i)_i\end{array}}{\Gamma\,[\Sigma]\!\!\vdash m\ \overline{n} \Rightarrow B}$$

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

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

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

T-Data
$$\frac{k\ \overline{A} \in D\ \overline{R} \qquad (\Gamma\,[\Sigma]\!\!\vdash n_i : A_i)_i}{\Gamma\,[\Sigma]\!\!\vdash k\ \overline{n} : D\ \overline{R}}$$

T-Command
$$\frac{c : \forall\overline{Z}.\overline{A \to}\ B \in \Sigma \qquad (\Gamma\,[\Sigma]\!\!\vdash n_i : A_i[\overline{R}/\overline{Z}])_i}{\Gamma\,[\Sigma]\!\!\vdash c\ \overline{R}\ \overline{n} : B[\overline{R}/\overline{Z}]}$$

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

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

T-LetRec
$$\frac{\begin{array}{c}\overline{P = \forall\overline{Z}.\{C\}} \\ \Gamma, \overline{f : P} \vdash \overline{e} : C \qquad \Gamma, \overline{f : P}\,[\Sigma]\!\!\vdash n : B\end{array}}{\Gamma\,[\Sigma]\!\!\vdash \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n : B}$$

T-Adapt
$$\frac{\Theta(\Sigma) = \Sigma' \qquad \Gamma\,[\Sigma']\!\!\vdash n : A}{\Gamma\,[\Sigma]\!\!\vdash \langle\Theta\rangle\ n : A}$$

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

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

$\boxed{r : T \dashv\![\Sigma]\ \Gamma}$

P-Value
$$\frac{\begin{array}{c}\Delta(\Sigma)\ \text{defined} \\ p : A \dashv \Gamma\end{array}}{p : \langle\Delta\rangle A \dashv\![\Sigma]\ \Gamma}$$

P-CatchAll
$$\frac{\Delta(\Sigma) = \Sigma'}{\langle x\rangle : \langle\Delta\rangle A \dashv\![\Sigma]\ x : \{[\Sigma']A\}}$$

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

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

P-Var
$$\frac{}{x : A \dashv x : A}$$

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

Fig. 3. Typing Rules

$\Gamma \vdash e : C$ states that in type environment Γ, computation e has type C. The judgement $r : T \dashv[\Sigma]\ \Gamma$ states that computation pattern r of argument type T with ambient ability Σ binds type environment Γ. The judgement $p : A \dashv \Gamma$ states that value pattern p of type A binds type environment Γ.

The T-Var rule infers the type of a monomorphic variable x by looking it up in the environment; T-PolyVar does the same for a polymorphic variable instantiation $f\ \overline{R}$, but also substitutes $\overline{R}$ for $\overline{Z}$ in its type. The T-App rule infers the type of an application $m\ \overline{n}$ under ambient ability Σ. First it infers the type of m of the form $\{\overline{\langle\Delta\rangle A \to} [\Sigma']B\}$. Then it checks that $\Sigma' = \Sigma$ and that each argument n_i matches the inferred type in the ambient ability Σ extended with adjustment Δ_i. If these checks succeed, then the inferred type for the application is B. The T-Ascribe rule ascribes a type to a construction allowing it to be treated as a use. Conversely, the T-Switch rule allows us to treat a use as a construction. The checking rules for data types (T-Data), commands (T-Command), suspended computations (T-Thunk), polymorphic let (T-Let), mutual recursion (T-LetRec), and adaptors (T-Adapt) recursively check the subterms. The T-Command rule looks up the type of the command c in the ambient ability. The T-App and T-Adapt rules have side-conditions on the computed abilities to ensure that they are well-defined (recall that the action of an adaptor / adjustment on an ability is a partial operation).

A computation of type $\overline{T \to} G$ is built by composing pattern matching clauses of the form $\overline{r} \mapsto n$ (Comp), where $\overline{r}$ is a sequence of computation patterns whose variables are bound in n. The side condition in the Comp rule requires that the patterns in the clauses cover all possible values inhabiting the argument types.

Value patterns match variables (P-Var) and data type constructor applications (P-Data). Value patterns can be typed as computation patterns (P-Value). To check a computation pattern $\langle x\rangle$ we apply the adjustment to the ambient ability (P-CatchAll). A command pattern $\langle c\ \overline{p} \to z\rangle$ may be checked at type $\langle\Delta\rangle B'$ with ambient ability Σ (P-Command). The command c must appear in the extension of Δ. The continuation is a plain function so its argument type has the identity adjustment. The continuation's return type has the ambient ability with Δ applied.

5.3 Operational Semantics

Runtime syntax. The operational semantics relies on the runtime syntax defined in Fig. 4. We distinguish between use values and construction values. Moreover, we also distinguish those construction values which are not uses, as these are the ones that can appear in type ascriptions in use values. We define a special class of *normal forms*, which are either construction values (w) or *frozen* evaluation contexts plugged with commands ($\lceil\mathcal{E}[c\ \overline{R}\ \overline{w}]\rceil$), described below. Normal forms can be regarded as generalised values that may be passed to operators.

Evaluation contexts are defined as sequences of evaluation frames. Evaluation frames are mostly unsurprising. The interesting case is $u\ (\overline{t}, [\ ], \overline{n})$ which enables evaluation to proceed on arguments left-to-right until each argument becomes a normal form. Following the separation between uses and constructions,

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

Fig. 4. Runtime Syntax

$\boxed{\Gamma\ [\Sigma]\text{-}\ m \Rightarrow A}$ $\boxed{\Gamma\ [\Sigma]\text{-}\ n : A}$

$$\textsc{T-Freeze-Use}\ \frac{\mathsf{level}(c, \mathcal{E}) > \bot \qquad \Gamma\ [\Sigma]\text{-}\ \mathcal{E}[c\ \overline{R}\ \overline{w}] \Rightarrow A}{\Gamma\ [\Sigma]\text{-}\ \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil \Rightarrow A} \qquad \textsc{T-Freeze-Cons}\ \frac{\mathsf{level}(c, \mathcal{E}) > \bot \qquad \Gamma\ [\Sigma]\text{-}\ \mathcal{E}[c\ \overline{R}\ \overline{w}] : A}{\Gamma\ [\Sigma]\text{-}\ \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil : A}$$

Fig. 5. Frozen Terms

evaluation contexts also ramify as such, leading to four distinct classes; for each possible combination of hole and body (see the R-Lift-$\star$ rules in Fig. 6). However, we do not explicitly distinguish the four kinds of evaluation context in the syntax but instead rely on the type system to do so.

Freezing commands. In order to handle a command we need to capture its delimited continuation, that is, the largest enclosing evaluation context that does not handle it. A frozen command $\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil$ is a command application $c\ \overline{R}\ \overline{w}$ plugged inside an evaluation context $\mathcal{E}$ that does not handle c. Commands may only be frozen at runtime. The typing rules for frozen commands are given in Fig. 5. Evaluation contexts can either be uses or constructions; hence we need two typing rules. The side-conditions depend on an auxiliary function: $\mathsf{level}(c, \mathcal{E})$, which is equal to the bottom element $\bot$, if c is handled by $\mathcal{E}$; or the required nesting depth of c handlers outside $\mathcal{E}$ to handle c, otherwise. The bottom element $\bot$ satisfies the following equations: $\forall i \neq \bot . \bot < i$ and $\forall i . \bot + i = \bot$.

Levels. The level of a command in an evaluation context is defined in terms of the level of a command in an evaluation frame.

$$\mathsf{level}(c, [\,]) = 0 \qquad\qquad \mathsf{level}(c, \mathcal{F}[\mathcal{E}]) = \mathsf{level}(c, \mathcal{F}) + \mathsf{level}(c, \mathcal{E})$$

The level of a command in an evaluation frame is 0 for all frames except argument frames and adaptor frames for which it is defined as follows.

$$\begin{array}{l}
\mathsf{level}(c, u\ (\overline{t}, [\,], \overline{n})) = \begin{cases} \bot, & \text{if } c \in \Xi \\ \mathsf{level}(c, \Theta_l), & \text{if } c \notin \Xi \end{cases} \\
\qquad\qquad\qquad\qquad \text{where } |\overline{t}| = l \wedge u = \uparrow(v : \{\overline{\langle \Theta | \Xi \rangle A \to}\ [\Sigma] B\}) \\
\mathsf{level}(c, \langle \Theta \rangle\ [\,]) = \mathsf{level}(c, \Theta)
\end{array}$$

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

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

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

R-Adapt
$$\frac{}{\langle \overline{\Theta} \rangle\ w \leadsto_{\mathrm{c}} w}$$

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

R-Freeze-Frame
$$\frac{\mathsf{level}(c, \mathcal{F}) > \bot\ \vee\ \mathsf{level}(c, \mathcal{E}) > 0}{\mathcal{F}[\lceil \mathcal{E}[c\ \overline{R}\ \overline{w}] \rceil] \leadsto_{\mathrm{c}} \lceil \mathcal{F}[\mathcal{E}[c\ \overline{R}\ \overline{w}]] \rceil}$$

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

Fig. 6. Operational Semantics

A handled command has level $\bot$. Otherwise the level is determined by the adaptor. The level of an adaptor is defined by the adaptor function bound to the interface that contains the command c.

$$\mathsf{level}(c, \Theta) = \begin{cases} \mathsf{level}(S \to S'), & \text{if } c \in I \wedge I(S \to S') \in \Theta \\ 0, & \text{if } c \in I \wedge I \notin \Theta \end{cases}$$

The level of an adaptor function is given by the level of the last component of the body in the pattern.

$$\mathsf{level}(S \to s) = \mathsf{level}(s, S) \qquad\qquad \mathsf{level}(S \to S'\ x) = \mathsf{level}(x, S)$$

Finally, the level of a variable in a pattern is its position in the pattern counting from right-to-left.

$$\mathsf{level}(s, s) = 0 \qquad \mathsf{level}(x, S\ x) = 0 \qquad \mathsf{level}(x, S\ y) = 1 + \mathsf{level}(x, S), \quad \text{if } x \neq y$$

The semantics is given in Fig. 6. We define four reduction relations: top-level use reduction ($m \leadsto_{\mathrm{u}} m'$), top-level construction reduction ($n \leadsto_{\mathrm{c}} n'$), full use reduction ($m \longrightarrow_{\mathrm{u}} m'$), and full construction reduction ($n \longrightarrow_{\mathrm{c}} n'$). We write $n[m/x]$ for n with m substituted for x and $n[\overline{m}/\overline{x}]$ for n with each m_i simultaneously substituted for each x_i. Similarly, we write $n[(n' : P)/f]$ for n with $(n' : P(\overline{R}))$ substituted for $f\ \overline{R}$ and the corresponding generalisation for simultaneous substitution (writing $P(\overline{R})$ for $A[\overline{R}/\overline{Z}]$ where $P = \forall \overline{Z}.A$).

$$\boxed{r : T \leftarrow t \mathrel{-\!\![\Sigma]}\ \theta}$$

$$\textsc{B-Value}\quad \frac{\Delta(\Sigma)\ \text{defined} \qquad p : A \leftarrow w \dashv \theta}{p : \langle\Delta\rangle A \leftarrow w \mathrel{-\!\![\Sigma]}\ \theta} \qquad\qquad \textsc{B-CatchAll}\quad \frac{\Delta(\Sigma) = \Sigma'}{\langle x\rangle : \langle\Delta\rangle A \leftarrow t \mathrel{-\!\![\Sigma]}\ [(\{t\} : \{[\Sigma']A\})/x]}$$

$$\textsc{B-Request}\quad \frac{\Delta = \Theta|\Xi \qquad \begin{array}{c}\Delta(\Sigma) = \Sigma' \qquad \mathsf{level}(c, \mathcal{E}) = 0\\ c : \forall \overline{Z}.\overline{B \to}\ B' \in \Xi\end{array} \qquad (p_i : B_i \leftarrow w_i \dashv \theta_i)_i}{\langle c\ \overline{p} \to z\rangle : \langle\Delta\rangle A \leftarrow \lceil \mathcal{E}[c\ \overline{R}\ \overline{w}]\rceil \mathrel{-\!\![\Sigma]}\ \overline{\theta}[(\{x \mapsto \mathcal{E}[x]\} : \{B' \to [\Sigma']A\})/z]}$$

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

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

Fig. 7. Pattern Binding

The R-Handle rule depends on two auxiliary judgements (Figure 7). The judgement $r : T \leftarrow u \mathrel{-\!\![\Sigma]}\ \theta$ states that computation pattern r of argument type T at ability Σ matches normal form t yielding substitution θ. The judgement $p : A \leftarrow w \mathrel{-\!\![\Sigma]}\ \theta$ states that value pattern p of type A at ability Σ matches construction value w yielding substitution θ. The R-Handle rule reduces an application according to a first-match semantics. The R-Ascribe-Use and R-Ascribe-Cons rules remove redundant type ascriptions from use and construction values respectively. The R-Let and R-LetRec rules are standard. The R-Adapt rule removes an adaptor once the construction it contains is a value. The R-Freeze-Comm rule freezes a command in the identity evaluation context. The R-Freeze-Frame rule extends the evaluation context of a frozen command by one frame. The R-Lift-UU, R-Lift-UC, R-Lift-CU, and R-Lift-CC rules lift the top-level reduction relations to full reduction relations.

Reduction preserves typing.

Theorem 1 (Subject Reduction).

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

Programs are constructions. If a program stops reducing then it must be a normal form, that is, either a value or a frozen evaluation context plugged with a command (and the latter only if the ambient ability is non-empty).

Definition 1. *We say that normal form t respects Σ if it is either a value w or a frozen evaluation context plugged with a command $\lceil\mathcal{E}[c\ \overline{R}\ \overline{w}]\rceil$ such that $c \in I$ and $\bot < \mathsf{level}(c, \mathcal{E}) < |\Sigma@I|$.*

Theorem 2 (Type Soundness).
If $\cdot\ [\Sigma]\text{-}\ n : A$ then either n is a normal form such that n respects Σ or there exists $\cdot\ [\Sigma]\text{-}\ n' : A$ such that $n \longrightarrow_{\mathrm{c}} n'$. (In particular, if $\Sigma = \emptyset$ then either n is a value w or there exists $\cdot\ [\Sigma]\text{-}\ n' : A$ such that $n \longrightarrow_{\mathrm{c}} n'$.)

6 Variations and Extensions

Adaptors are one particular solution to the effect pollution problem and we have detailed one particular design for adaptors. In this section we discuss variations and extensions of adaptors.

Adaptors Before Extensions. As spelled out in Sect. 5, the action of an adjustment on an ability first applies the adaptor and then applies the extension. If we were to switch the order then we could obtain a slightly more expressive system in which one could handle an instance of an interface other than the rightmost one. For example an argument type `<Abort(s x y -> s y x)|Abort>Int` would handle the second instance of `Abort` rather than the first as it does now. A disadvantage of switching the order is that common cases such as mask become a little more complicated.

Deep Extensions. We have implemented an extension to adaptors in Frank whereby multiple instances of the same effect interface can be handled by the same effect handler. We distinguish the instances by indexing by position in command patterns. For instance, the following handler is able to handle `Reader Bool` and `Reader Int` effects at the same time.

```
readTwo : {<Reader Bool, Reader Int>X -> X}
readTwo { x              -> x
        | <ask.0 -> r> -> r 42
        | <ask.1 -> r> -> r true }
```

The usual `ask` in a command pattern is syntactic sugar for `ask.0`. We might further extend this syntax with a way of naming handled effect instances.

Extensible Adaptors. Another design that may be worth investigating would be to roll extensions into adaptors. For instance, `Reader(s -> s (Int))` would indicate the adjustment `|Reader Int` and `Reader(s x -> s (Int) x)` an adjustment that handles the second instance of `Reader`.

Adaptive Abilities. A nonuniformity in our current design is that both adaptors and extensions appear in adjustments yet only extensions appear in abilities. One obtains a more expressive system by changing the syntax of abilities to:

$$\Sigma ::= \sigma|\Delta$$

The adaptor in an ability applies after the effect variable has been instantiated. This requires some care because there is no guarantee that an adaptor pattern will match. A solution is to change the semantics of pattern matching such that variables are always bound to (possibly empty) lists of instances and the body of an adaptor component concatenates all of the lists together. For instance, suppose we apply the adaptor `Reader(s x y -> s y)` to the ability `0|Reader Int`, then `y` is bound to the singleton list containing the one instance of `Reader Int` and `x` and `s` are both bound to the empty list. With adaptive abilities one can define an adaptor that disables all instances of an interface. For instance, we could disable all instances of `Console` with the adaptor: `Console(s ->)`.

Inference. One might hope to automatically infer which inline adaptors need to be inserted in order to make a program type check. This is possible up to a point, but clearly not in general as there cannot be a unique or most general solution. For instance, if we need to coerce a computation with ability `Reader Int` to one with ability `Reader Int, Reader Int`, then we must choose between two possible adaptors: `Reader(s x -> s)` and `Reader(s x y -> s y)`.

7 Related Work

Effect Instances. Brady [5] provides a lightweight syntax for statically supporting multiple instances of the same effect. Early versions of the Eff programming language [1,2] include a facility to dynamically generate fresh effect instances offering an alternative way to encapsulate effects. We could consider adding a fresh effect construct to Frank. In order to do so we would need to extend the effect type system to provide interface variables (along the lines of a region type system) and this may admit programming patterns that are awkward or impossible with only adaptors. On the other hand fresh effect generation is not expressive enough to define mask, so we would probably still want to keep adaptors. In contrast, for row-based effect type system that do not allow effect shadowing, such as the one employed by Links [9], a fresh effect construct would suffice, as there is no need for mask anyway.

Inject, Lift, and Mask. Inject, lift, and mask are different names for the same construct. Leijen's Koka language [13] has a similar row-based effect type system to Frank in which effects can shadow one another. When Koka was originally conceived it had no effect handlers. Nevertheless, Leijen discusses inject [13] as a way to address a particular instance of the effect pollution problem for hard-coded exceptions. Koka now supports effect handlers [14] and a generalised inject for arbitrary effects [15]. Biernacki et al. [3] introduce an effect handler calculus $\lambda^{H/L}$, inspired by Koka. They construct a logical relation which they use for reasoning about effect handler programs. Inspired by a desire for greater parametricity, they incorporate lift. They also observe that they can macro-express a generalised lift operator and a swap operator using plain lift and effect handlers. Extending $\lambda^{H/L}$, Biernacki et al. [4] introduce a calculus λ^{HEL} combining existential types, effect instances with a region-like type system, and a generalisation of lift similar to adaptors that they call *coercions.* Unlike adaptors which are expressed as lambda expressions, coercions are constructed from four basic combinators: lift, swap, cons, and compose. These coercions are not quite as expressive as adaptors as they do not support duplicating an effect (as in our state actors example), though one can work around this limitation using a handler.

Modules. Multicore OCaml [8] supports effect handlers without effect types. The OCaml module system provides a mechanism for simulating effect instances. Biernacki et al.'s programming language Helium [4] also incorporates a module system, which is elaborated into λ^{HEL} using existential types.

Negative Adjustments. In prior work [16] we proposed *negative adjustments* as a means for masking effects from operator arguments; these correspond exactly to the standard mask adaptor adjustment of our current design.

Effect Tunneling. Zhang and Myers [25] present an alternative approach to ensure handlers encapsulate effects. Their key idea is to extend types with *handler variables* and handler polymorphism. The variables act as labels on effects in the ambient context and determine which handlers interpret which effects. Handler polymorphism is resolved by substituting the nearest lexically enclosing handler for the handler variable. A capability region system ensures that computations do not escape their handlers. Following Biernacki et al. [3], they develop a sound logical relations model and prove their system satisfies an abstraction theorem. While Zhang and Myers do not present an implementation, they claim that the programmer need not deal with handler variables in practice. Rather, they outline a desugaring and rewriting pass which inserts the requisite variable bindings and handler instantiations.

8 Conclusion

We have studied how to encapsulate effects in Frank and avoid the effect pollution problem which manifests when composing operators whose intermediate effects overlap with external effects. We addressed the problem using adaptors which allow a flexible reconfiguration of effects and integrate smoothly in the static and dynamic semantics of Frank. We have demonstrated the practical importance of effect encapsulation through an implementation of actors, which would egregiously leak its implementation details if written naively without adaptors, but which keeps them completely hidden with judicious use of adaptors. We extended the implementation of Frank with adaptors and other features needed for our actor case study such as polymorphic commands and ML-style references. Finally, we have presented a formal type system and semantics for Frank and proved type soundness.

References

1. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. In: CALCO. Lecture Notes in Computer Science, vol. 8089, pp. 1–16. Springer (2013)
2. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebr. Meth. Program. **84**(1), 108–123 (2015)
3. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Handle with care: relational interpretation of algebraic effects and handlers. PACMPL **2**(POPL) (2018)
4. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Abstracting algebraic effects. PACMPL **3**(POPL) (2019)
5. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ICFP. pp. 133–144. ACM (2013)
6. Convent, L.: Enhancing a modular effectful progamming language. M.Sc. dissertation, University of Edinburgh (2017)
7. Convent, L., Lindley, S., McBride, C., McLaughlin, C.: Frank adaptors branch (2018), `https://github.com/frank-lang/frank/tree/adaptors`
8. Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J., Madhavapeddy, A.: Effective concurrency through algebraic effects. In: OCaml Workshop (2015)
9. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: TyDe@ICFP. pp. 15–27. ACM (2016)
10. Hillerström, D., Lindley, S.: Shallow effect handlers. In: APLAS. Lecture Notes in Computer Science, vol. 11275. Springer (2018)
11. Huet, G.P.: The zipper. J. Funct. Program. **7**(5), 549–554 (1997)
12. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: ICFP. pp. 145–158. ACM (2013)
13. Leijen, D.: Koka: Programming with row polymorphic effect types. In: MSFP. EPTCS, vol. 153, pp. 100–126 (2014)
14. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: POPL. pp. 486–499. ACM (2017)
15. Leijen, D.: Algebraic effect handlers with resources and deep finalization. Tech. rep., Microsoft Research (2018)
16. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: POPL. pp. 500–514. ACM (2017)
17. Moggi, E.: Computational lambda-calculus and monads. In: LICS. pp. 14–23. IEEE Computer Society (1989)
18. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. **22**(1), 1–44 (2000)
19. Piróg, M., Schrijvers, T., Wu, N., Jaskelioff, M.: Syntax and semantics for operations with scopes. In: LICS. pp. 809–818. ACM (2018)
20. Plotkin, G.D., Power, J.: Adequacy for algebraic effects. In: FoSSaCS. Lecture Notes in Computer Science, vol. 2030, pp. 1–24. Springer (2001)
21. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: FoSSaCS. Lecture Notes in Computer Science, vol. 2303, pp. 342–356. Springer (2002)
22. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Applied Categorical Structures **11**(1), 69–94 (2003)
23. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science **9**(4) (2013)
24. Wadler, P.: Comprehending monads. In: LISP and Functional Programming. pp. 61–78 (1990)
25. Zhang, Y., Myers, A.C.: Abstraction-safe effect handlers via tunneling. PACMPL **3**(POPL), 5:1–5:29 (2019)