JFP **30**, e9, 53 pages, 2020.
doi:10.1017/S0956796820000039

Doo bee doo bee doo

LUKAS CONVENT
University of Lübeck, Lübeck, Germany
(*e-mail:* convent@isp.uni-luebeck.de)

SAM LINDLEY
The University of Edinburgh, Edinburgh EH8 9YL, UK and
Imperial College London,
London SW7 2BU, UK
(*e-mail:* sam.lindley@ed.ac.uk)

CONOR MCBRIDE
University of Strathclyde, Glasgow G1 1XQ, UK
(*e-mail:* conor.mcbride@strath.ac.uk)

CRAIG MCLAUGHLIN
The University of Edinburgh, Edinburgh EH8 9YL, UK and University of New South Wales, Kensington, Australia and *Data61, CSIRO, Australia*
(*e-mail:* c.mclaughlin@unsw.edu.au)

Abstract

We explore the design and implementation of Frank, a strict functional programming language with a bidirectional effect type system designed from the ground up around a novel variant of Plotkin and Pretnar's effect handler abstraction. Effect handlers provide an abstraction for modular effectful programming: a handler acts as an interpreter for a collection of commands whose interfaces are statically tracked by the type system. However, Frank eliminates the need for an additional effect handling construct by generalising the basic mechanism of functional abstraction itself. A function is but the special case of a Frank *operator* that interprets no commands. Moreover, Frank's operators can be *multihandlers* which simultaneously interpret commands from several sources at once, without disturbing the direct style of functional programming with values. Effect typing in Frank employs a novel form of effect polymorphism which avoids mentioning effect variables in source code. This is achieved by propagating an *ambient ability* inwards, rather than accumulating unions of potential effects outwards. With the ambient ability describing the effects that are available at a certain point in the code, it can become necessary to reconfigure access to the ambient ability. A primary goal is to be able to encapsulate internal effects, eliminating a phenomenon we call *effect pollution*. Moreover, it is sometimes desirable to rewire the effect flow between effectful library components. We propose adaptors as a means for supporting both effect encapsulation and more general rewiring. Programming with effects and handlers is in its infancy. We contribute an exploration of future possibilities, particularly in combination with other forms of rich type systems.

1 Introduction

Shall I be pure or impure?

—Philip Wadler (1992)

We say "Yes": purity is a choice to make *locally*. We introduce **Frank**, an applicative language where the meaning of "impure" computations is open to negotiation, based on Plotkin and Pretnar's effect handlers (Plotkin & Pretnar, 2013)—a rich foundation for effectful programming. By separating effect interfaces from their implementation, effect handlers offer a high degree of modularity. Programmers can express effectful programs independently of the concrete interpretation of their effects. A handler gives one interpretation of the effects of a computation. In Frank, effect types (sometimes called simply *effects* in the literature) are known as *abilities*. An ability denotes the permission to invoke a particular set of commands.

Frank programs are written in direct style in the spirit of effect type systems (Lucassen & Gifford, 1988; Talpin & Jouvelot, 1994). Frank *operators* generalise call-by-value functions in two dimensions. First, operators handle effects. A unary operator is an effect handler, acting as an interpreter for a specified set of commands whose types are statically tracked by the type system. A unary function is but the special case of a unary operator whose handled command set is empty. Second, operators are n-ary, handling multiple computations over distinct command sets simultaneously. An n-ary function is but the special case of an n-ary operator whose handled command sets are all empty.

The contributions of this paper are:

- the definition of Frank, a strict functional programming language featuring a bidirectional effect type system, effect polymorphism, and effect handlers;
- operators as both *multihandlers* for handling multiple computations over distinct effect sets simultaneously and as *functions* acting on values;
- a novel approach to effect polymorphism which avoids mentioning effect variables in source code, crucially relying on the observation that one must always instantiate the effects of an operator being applied with the *ambient ability*, that is, precisely those effects permitted by the current typing context;
- a small-step operational semantics for Frank and a proof of type soundness;
- a flexible way of reconfiguring access to the ambient ability via *adaptors*, thereby eliminating the phenomenon of effect pollution and facilitating composition of effectful components;
- an extended example making essential use of adaptors in composing handlers together to implement concurrent actors with mailboxes;
- an exploration of directions for future research, combining effect-and-handlers programming with features including substructural typing, dependent types, and totality.

A number of other languages and libraries are built around effect handlers. Bauer and Pretnar's Eff (Bauer & Pretnar, 2015) language is an ML-like language extended with effect handlers. A significant difference between Frank and the original version of Eff is that the latter provides no support for effect typing. Recently, Bauer and Pretnar

have designed an effect type system for Eff (Bauer & Pretnar, 2014). Their implementation (Pretnar, 2014) supports Hindley–Milner type inference, and the type system incorporates effect subtyping. Helium (Biernacki *et al.*, 2019) is a more recent experimental ML-like language with support for effect handlers, which incorporates facilities for effect encapsulation including a mechanism for generating fresh effects and a module system that supports abstract effects.

Hillerström and Lindley (Hillerström, 2015, 2016; Hillerström & Lindley, 2016; Hillerström *et al.*, 2020) in Links (Cooper *et al.*, 2006) and Leijen (2017) in Koka (Leijen, 2014) have extended existing languages with effect handlers. Both languages incorporate row-based effect type systems and attempt to elide some effect variables from source code, but neither eliminates effect variables to the extent that Frank does. Dolan *et al.* (2015) built Multicore OCaml by extending OCaml with support for effect handlers. Multicore OCaml does not include an effect type system.

Whereas Frank is bidirectionally typed, all of these other languages use Hindley–Milner type inference. None of the other languages supports multihandlers and none of those with effect typing allows effect variables to be omitted to the degree that Frank does.

Kammar *et al.* (2013) describe a number of effect handler libraries for languages ranging from Racket to SML, to OCaml, and to Haskell. Apart from the Haskell library, their libraries have no effect typing support. The Haskell library takes advantage of type classes to simulate an effect type system not entirely dissimilar to that of Frank. As Haskell is lazy, the Haskell library cannot be used to write direct-style effectful programs—one must instead adopt a monadic style. Moreover, although there are a number of ways of almost simulating effect type systems in Haskell, none is without its flaws. Kiselyov and collaborators (Kiselyov *et al.*, 2013; Kiselyov & Ishii, 2015) have built another Haskell library for effect handlers, making different design choices.

Recently, there have been several efforts to build object-oriented effect handler libraries in Scala (Brachthäuser *et al.*, to appear; Brachthäuser & Schuster, 2017) and Java (Brachthäuser *et al.*, 2018; Inostroza & van der Storm, 2018). These incorporate effect type systems not dissimilar to those of Kammar *et al.*'s Haskell library, encoded via Java classes in place of type classes.

Brady's `effects` library (Brady, 2013b) provides a domain specific language for programming with effects in the dependently typed language Idris (Brady, 2013a). Like the Haskell libraries, Brady's library currently requires the programmer to write effectful code in a monadic style.

McBride's "shonky" (McBride, 2016) is essentially an untyped version of Frank, with a somewhat different concrete syntax. Our implementation of Frank translates typed Frank programs into shonky. The implementation of Frank is available at the following URL:

https://www.github.com/frank-lang/frank

The rest of the paper is structured as follows. Section 2 introduces Frank by example. Section 3 motivates adaptors in Frank by example. Section 4 presents the syntax, type system, and small-step operational semantics for Frank accompanied with a proof of type soundness. Section 5 discusses how to store computations in data structures. Section 6 presents an extended example making essential use of adaptors in composing

handlers together to implement concurrent actors with mailboxes. Section 7 describes our implementation. Section 8 discusses adaptor variations and extensions. Section 9 outlines related work. Section 10 discusses future work and Section 11 concludes.

Relation to Prior Work. This is an updated and extended version of a previous paper (Lindley *et al.*, 2017). The primary new contributions in this paper are a direct semantics (previously the semantics was given by translation to a core language) and extensions to Frank with *polymorphic commands* and *adaptors* to support effect encapsulation and enable the rewiring of effect flow. We have changed the title to include three letter words in the hope that Google Scholar will index it this time.

2 A Frank tutorial

> "To be is to do"—Socrates.
> "To do is to be"—Sartre.
> "Do be do be do"—Sinatra.
> —anonymous graffiti, via Kurt Vonnegut (1982)

Frank is a functional programming language with effects and handlers in the spirit of Eff (Bauer & Pretnar, 2015), controlled by a type system inspired by Levy's call-by-push-value (CBPV) (Levy, 2004). Doing and Being are clearly separated and managed by distinguished notions of computation and value types.

2.1 Data types and first-order functions

Concrete values live in inductive data types. By convention (not compulsion), we give type constructors uppercase initials and may apply prefixed to parameters, also written uppercase. Data constructors are prefix and, again by convention, initially lowercase.

```
data Zero =
data Unit = unit
data Bool = tt | ff

data Nat    = zero | suc Nat
data List X = nil  | cons X (List X)

data Pair X Y = pair X Y
```

The data types `Int` and `Char` are built in and `String` is an alias for `List Char`. There is syntactic sugar for lists, for instance:

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

We choose to treat constructors as distinct from functions, and constructors must always be fully applied.

We can write perfectly ordinary first-order functional programs by pattern matching. Type signatures are compulsory, universally quantifying implicitly over freely occurring type variables and insisting on the corresponding parametric polymorphism.

```
append : {List X -> List X -> List X}
append nil      ys = ys
append (x::xs) ys = x::(append xs ys)
```

2.2 Effect polymorphism in ambient silence

Computations, such as functions, have computation types, which embed explicitly into the value types: braces play the role of "suspenders" in types and values. Accordingly, we can write typical higher-order functions

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

and apply them in the usual way:

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

A value type `A` is a data type `D` $\texttt{R}_1$ ... $\texttt{R}_n$, a suspended computation type `{C}`, or a type variable `X`. A computation type $\texttt{T}_1$ `->` ... `->` $\texttt{T}_m$ `->` `[`$\texttt{I}_1$`, ...,` $\texttt{I}_n$`]B` resembles a function type with m argument types and a return type showing the *ability* the computation needs—a bracketed list of n *interface instances*—and the *value type* it delivers. In Frank, names always bind values (a simplifying decision which we shall re-examine in Section 10). Top-level definitions give names to suspended computations.

Typechecking separates cleanly into checking the compatibility of value types and checking that required abilities are available. Empty abilities may be omitted but take a care as this does not denote purity. Instead, the definition

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

is really a short hand for

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

which makes the `map` operator implicitly effect-polymorphic in the effect variable ε. The type of `map` in Frank says that whatever ability an instance receives will be offered in turn to the operator that acts on each element. That is, we have written something like ML's `map` but without giving up control over effects, and we have written something like Haskell's `map` but acquired a function as general as its monadic `mapM`, as we shall see just as soon as we acquire nontrivial ability.

2.3 Controlling evaluation

Frank is a (left-to-right) call-by-value language, so we should be careful when defining control operators. For instance, we may define sequential composition operators

```
fst : {X -> Y -> X}          snd : {X -> Y -> Y}
fst x y = x                  snd x y = y
```

Both arguments are evaluated (relative to the ambient ability), before one value is returned. We take the liberty of writing `snd x y` as `x; y`, echoing the ML semicolon, and note its associativity.

Meanwhile, avoiding evaluation must be done explicitly by suspending computations. The following operator:

```
iffy : {Bool -> X -> X -> X}
iffy tt t f = t
iffy ff t f = f
```

is the conditional expression operator which forces evaluation of the condition and *both* branches, before choosing between the values. To write the traditional conditional, we must therefore suspend the second and third arguments:

```
if : {Bool -> {X} -> {X} -> X}
if tt t f = t!
if ff t f = f!
```

Again, Frank variables stand for values, but `t` and `f` are not values of type `X`. Rather, they *are* suspended computations of type `{X}`, but we must *do* just one. The postfix `!` denotes nullary application of a suspended computation.

We write suspended computations in braces, with a choice of zero or more pattern matching clauses separated by `|` symbols. In a nullary suspension, we have one choice, which is just written as an expression in braces, for instance,

```
if fire! {launch missiles} {unit}
```

assuming that `launch` is a command permitted by the ambient ability, granted to both branches by the silently effect-polymorphic type of `if`.

With non-nullary suspensions, we can define case-expressions inline using reverse application

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

as in this example of the short-circuited “and”:

```
shortAnd : {Bool -> {Bool} -> Bool}
shortAnd x c = case x { tt -> c! | ff -> ff }
```

2.4 Abilities collect interfaces; Interfaces offer commands

Abilities (Frank’s realisation of effects) are collections of instantiated interfaces, each of which describes a choice of commands (known elsewhere as *operations*; Plotkin & Pretnar, 2013). Here are some simple interface definitions.

```
interface Send X     = send : X -> Unit

interface Receive X  = receive : X

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

The `send` command takes an argument of type `X` and returns a value of type `Unit`. The `receive` command returns a value of type `X`. The `State` interface offers `get` and `put` commands. Note that, unlike data constructors, commands are first-class values. In particular, while `Zero` is uninhabited, `{[Receive Int]Zero}` contains the value `receive!`.

We may use the silent effect polymorphism of `map` nontrivially to send a list of elements, one at a time:

```
sends : {List X -> [Send X]Unit}
sends xs = map send xs; unit
```

The reason this type checks at all is because `map` is implicitly polymorphic in its effects. The bracket `[Send X]` demands that the ambient ability permits *at least* the `Send X` commands.

The type of `map` works with *any* ambient ability, hence certainly with those permitting `Send X`, and it passes that ability to its computation argument, which may thus be `send`.

However, the following does not typecheck, because `Send X` has not been included in the return type of `bad`.

```
bad : {List X -> Unit}
bad xs = map send xs; unit
```

There is no effect inference in Frank. The typing rules' conclusions do not accumulate the abilities of the programs in their premisses. Rather, we are explicit about what the environment makes possible—via the ambient ability—and where and how that changes.

In designing Frank, we have sought to maintain the benefits of effect polymorphism while avoiding the need to write effect variables in source code. There are no explicit effect variables in any of the examples in this paper. In an earlier version of Frank, we ruled out explicit effect variables by fiat. But doing so placed artificial restrictions on the formalism (see Section 4), so we do now permit them. A case where explicit effect variables may be useful is in manipulating data types containing multiple suspended computations with different abilities; we are yet to explore compelling use cases.

2.5 Polymorphic commands

The original version of Frank (Lindley *et al.*, 2017) supports only monomorphic commands. We have extended it with polymorphic commands (Kammar *et al.*, 2013) which can be invoked at any type.

```
interface Abort = abort X : X
```

The `abort` command is suitable for expressing abortion of a computation: it can be invoked anywhere as it produces a value of any type `X` and it is guaranteed to end computation as there is no way to produce a value for a generic `X`. Previously, we had to do some gymnastics to get the same effect.

```
interface Abort' = aborting : Zero

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

The postfix `!` attached to `abort` denotes the definition of a nullary operator. The `abort'` operator simulates the `abort` command by requesting a value of type `Zero`, which guarantees the end of computation as there is no way to produce such a value. Furthermore, polymorphic commands are necessary for more interesting examples like ML-style dynamically allocated references.

ML-style References. Based on polymorphic commands, 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 yet expressive enough to define them in the language. We give an example in Section 5 of how built-in effects are declared in the `main` program to be handled externally.

2.6 Direct style for monadic programming

We work in a direct applicative style. When the output of one computation is used as the input to another, we may just write an application, or a case analysis, directly. For instance, we can implement the result of repeatedly reading lists until one is empty and concatenating the result.

```
catter : {[Receive (List X)]List X}
catter! = case receive! { nil -> nil
                        | xs  -> append xs catter! }
```

In Haskell, `receive!` would be a monadic computation `ask` unsuitable for case analysis—its value would be extracted and named before inspection, thus:

```
catter :: Reader (List a) (List a)   -- Haskell
catter = do
  xs <- ask
  case xs of
    [] -> return []
    xs -> do ys <- catter; return (xs ++ ys)
```

The latter part of `catter` could perhaps be written without naming `ys` as `(xs ++)<$> catter`, or even, with "idiom brackets", `(|pure xs ++ catter|)`, but always there is extra plumbing (here `do`-notation and `return`) whose only purpose is to tell the compiler where to parse a type as *effect value* and where just as *value*. The choice to be frank about the separation of effects from values in the syntax of types provides a stronger cue to the status of each component and reduces the need for plumbing. We do not, however, escape the need to disambiguate *doing* `receive!` from *being* `receive`.

In the same mode, we can implement the `C++` "increment `c`, return original value" operation as follows:

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

In Haskell, `next` would have to be explicitly sequentialised.

```
next :: State Int Int
next = do x <- get
          y <- get
          put y+1
          return x
```

(We have written `get` twice to match the preceding Frank code, but assuming a standard implementation of state one could of course delete the second `get` and use `x` in place of `y`.) The absence of explicit plumbing in Frank depends crucially on the fact that Frank, unlike Haskell, has a fixed evaluation order.

2.7 Handling by application

In a call-by-value language, a function application can be seen as a particularly degenerate mode of coroutining between the function and its argument. The function process waits while the argument process computes to a value, transmitted once as the argument's terminal action; on receipt, the function post-processes that value in some way, before transmitting its value in turn.

An exception handler is also a means for one process to contextualise the execution of another, remaining entirely dormant and dismounting itself without fuss if its subordinate runs without failure, but stepping into the breach if needs be. Pascal's "with…do…" contextualises the "do" part by bringing into scope the fields of the record given by the "with" (Jensen & Wirth, 1974). One of the third author's early programming languages, from 1991, extends LOGO (Papert, 1980) with a first class and compositional notion of *context*, generalising functions, records, exception handlers, and localised remapping—handling, in today's terminology—of the "turtle graphics" commands, to support mathematical investigations by diagram transformation (McBride, 2011). Its syntax for contextualisation is exactly the blank space of application, and it is from that early experience that we take elements of our present design.

Frank is already distinct from other languages with effect handlers in its effect type system, but the key departure it makes in program style is to handle effects without any special syntax for invoking an effect handler. Rather, the ordinary notion of "function" is extended with the means to offer effects to arguments, invoked just by application. That is, the blank space signifies more general modes of coroutining between operator and arguments than the return-value-then-quit default. For instance, the usual behaviour of the "state" commands can be given as follows:

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

Let us give an example using `state` before unpacking its definition. We might pair the elements of a list with successive numbers.

```
index : {List X -> List (Pair Int X)}
index xs = state 0 (map {x -> pair next! x} xs)
```

Allowing string notation for lists of characters, we obtain:

```
index "abc" ⟹ [(pair 0 'a'), (pair 1 'b'), (pair 2 'c')]
```

What is happening?

The type of `state` shows us that Frank operators do not merely have argument value types but also an *adjustment* to the ambient ability, written in chevrons and usually omitted when it is the identity (as in all of the examples we have seen so far). Whatever the ambient ability might be when `state` is invoked, the initial state should arrive at its first argument using only that ability; the ambient ability at its second argument will include the `State S` interface, shadowing any other `State A` interfaces which might have been present already. Correspondingly, by the time `index` invokes `map`, the ambient ability includes `State Int`, allowing the elementwise operation to invoke `next!`.

The first equation of `state` explains what to do if any *value* is passed as the second argument. In Frank, a traditional pattern built from constructors and variables matches only *values*, so the `x` is not a catch-all pattern, as things other than values can be passed as that argument. In particular, *requests* can be passed as the second argument, in accordance with the `State S` interface. A request consists of a command instance and a continuation. Requests are matched by patterns in chevrons which show the particular command instance being handled left of `->`, with a pattern variable standing for the *continuation* on the right. The patterns of `state` thus cover all possible *signals* (that is, values or requests) advertised as acceptable at its arguments.

Having received signals for each argument the `state` operator should *handle* them. If the second argument is a value, then that value is returned. If the second argument is a request, then the `state` operator is reinvoked with a new state (which is simply the old state in the case of `get` and `s` in the case of `put s`) in the first argument and the continuation *invoked* in the second argument.

We emphasise a key difference between Frank and most other languages supporting effect handlers (including Eff, Helium, Koka, and Multicore OCaml Dolan *et al.*, 2015): Frank's continuation variables are shallow in that they capture only the rest of the subordinated computation, not the result of handling it, allowing us to change how we carry on handling, for instance, by updating the state. In contrast, Multicore OCaml's continuation variables are deep in that invoking them implicitly reinvokes the handler. Consider the definition for `state` in Multicore OCaml

```
effect Put : t -> unit
let put x = perform (Put x)

effect Get : t
let get () = perform Get

let state m =
  match m () with
  | x                -> fun s -> x
  | effect Get     k -> fun s -> continue k s s
  | effect (Put s) k -> fun _ -> continue k () s
```

Multicore OCaml provides three special keywords for effects and handlers: `effect` declares a command or marks a request pattern, `perform` invokes a command, and `continue` invokes a continuation. The shallow implementation of `state` in Frank requires explicit recursion. The deep implementation of `state` in Multicore OCaml performs the recursion implicitly. On the other hand, the shallow version allows us to thread the state through the operator, whereas the deep version relies on interpreting a stateful computation as a function and threading the state through the continuation.

Shallow handlers can straightforwardly express deep handlers using explicit recursion. Deep handlers can encode shallow handlers in much the same way that iteration (catamorphism, fold) can encode primitive recursion (paramorphism) and with much the same increase in complexity. On the other hand, handlers which admit a deep implementation have a more regular behaviour and admit easier reasoning, just as "folds" offer specific proof techniques not available to pattern matching programs in general. Kammar *et al.*

(2013) provide a more in-depth discussion of the trade-offs between deep and shallow handlers.

2.8 Handling multiple arguments at once

Frank allows the programmer to write *n*-ary operators, so we can offer different adjustments to the ambient ability at different arguments. For instance, we can implement a `pipe` operator which matches `receive` commands downstream with `send` commands upstream.

```
pipe : {<Send X>Unit -> <Receive X>Y -> [Abort]Y}
pipe <send x -> s> <receive -> r> = pipe (s unit) (r x)
pipe <_>           y              = y
pipe unit          <_>            = abort!
```

The type signature conveys several different things. The `pipe` operator must handle all commands from `Send X` on its first argument and all commands from `Receive X` on its second argument. We say that `pipe` is thus a *multihandler*. The value of the first argument has type `Unit` and the value of the second argument has type `Y`. The operator itself is allowed to perform `Abort` commands and returns a final value of type `Y`.

The first line implements the communication between producer and consumer, reinvoking `pipe` with both continuations, giving the sent value to the receiver. The second line makes use of the catch-all pattern `<_>` which matches *either* a `send` command or an attempt to return a value: as the consumer has delivered a value the producer can be safely discarded. The third line covers the case which falls through: the catch-all pattern must be a `receive` command, as the value case has been treated already, but the producer has stopped sending, so `abort` is invoked to indicate a "broken pipe".

We can run `pipe` as follows:

```
pipe (sends ["do", "be", ""])
     catter!
⟹ "dobe"
```

Moreover, if we write

```
spacer : {[Send (List Char), Receive (List Char)]Unit}
spacer! = send receive; send " "; spacer!
```

we find instead that

```
pipe (sends ["do", "be", ""])
     (pipe spacer! catter!)
⟹ "do be "
```

where the `spacer`'s `receive`s are handled by the outer `pipe`, but its `send`s are handled by the inner one. The other way around also works as it should, that is, `pipe` is associative.

```
pipe (pipe (sends ["do", "be", ""]) spacer!)
     catter!
⟹ "do be "
```

There is nothing you can do with simultaneous handling that you cannot also do with mutually recursive handlers for one process at a time. The Frank approach is, however, more direct. Kammar *et al.* (2013) provide both deep and shallow handler implementations

for pipes using their Haskell effects library. Both implementations are significantly more complex than the above definition in Frank, requiring the unary handler for sending (receiving) to maintain a suspended computation to the consumer (producer) to continue the interaction upon receipt of a command. Moreover, the deep handler implementation depends on a nontrivial mutually recursive data type, which places considerable cognitive load on the programmer. So, even in systems such as Multicore OCaml and Eff, offering a more aesthetic syntax than Kammar *et al.*'s library, a programming burden remains.

Let us clarify that the adjustment marked in chevrons in an argument type promises *exactly* what will be handled for that argument. The return type of `pipe` requires the ambient ability to support `Abort`, and its arguments offer to extend that ability with `Send X` and `Receive X`, respectively, so the producer and consumer will each also support `Abort`. However, because neither argument type advertises `Abort` in its adjustment, the implementation of `pipe` may not intercept the `aborting` command. In particular, the catch-all pattern `<_>` matches only the signals advertised by the relevant argument type, with other commands forwarded transparently to the most local argument offering the relevant interface. No Frank process may secretly intercept commands. Of course, the `pipe` operator can prevent action by ignoring the continuation to a `send` on its first argument or a `receive` on its second, but it cannot change the meaning of other commands.

One can simulate adjustments using a more conventional effect type system with abilities on both argument types and return types. However, this yields more verbose and less precise types. For instance, the type of the first argument to `pipe` becomes `<Abort, Send X>Unit` instead of `<Send X>Unit`. The argument type has been polluted by the ability of the return type and it now fails to advertise precisely which effects it handles and which effects it forwards untouched.

2.9 The catch question

Frank allows us to implement an "exception handler" with a slightly more nuanced type than is sometimes seen.

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

The first argument to `catch` is the computation to run that may raise an exception. The second argument is the alternative computation to run in the case of failure, given as a suspended computation allowing us to choose whether to run it. We do not presume that the ambient ability in which `catch` is executed offers the `Abort` interface. In contrast, a typical treatment of exceptions renders `catch` as the prioritised choice between two failure-prone computations. For instance, the Haskell mtl library offers

```
catchError :: -- Haskell
  MonadError () m => m a -> (() -> m a) -> m a
```

where the exception handler is always allowed to throw an error. In other words, this Haskell typing unnecessarily makes the ability to abort non-local. Leijen makes a similar observation in Koka's treatment of exceptions (Leijen, 2014).

Frank's effect polymorphism ensures that the alternative computation is permitted to abort if and only if `catch` is, so we lose no functionality but gain precision. Moreover, we promise that `catch` will trap `abort` only in its first argument, so that any failure (or anything else) that `h!` does is handled by the environment—indeed, you can see that `h!` is executed as a tail call, if at all, thus outside the scope of `catch`. In the case that the ambient ability is allowed to abort, then when the adjustment is applied to the ambient ability we obtain an ability with two instances of the `Abort` interface. Frank allows duplicate instances and respects their order, i.e. an ability is a map from interfaces to lists of interface instances. When invoking a command, it is always directed at the rightmost interface instance, i.e. the one most recently added.

2.10 The disappearance of control

Using one of the many variations on the theme of free monads, we could implement operators like `state`, `pipe`, and `catch` as abstractions over *computations* reified as command-response trees. By contrast, our handlers do not abstract over computations, nor do they have computation-to-computation handler types distinct from value-to-computation function types (Kammar *et al.*, 2013; Bauer & Pretnar, 2014).

Frank computations are abstract: a thing of type `{C}` can be communicated or invoked but not inspected. Argument types explain which values are expected, and operators match on those values directly, without apparently forcing a computation, yet they also admit other specific modes of interaction, handled in specific ways.

Semantically, then, a Frank operator must map computation trees to computation trees, but we write its action on values directly and its handling of commands minimally. The machinery by which commands from the input not handled locally must be forwarded with suitably wrapped continuations is hard-wired, as we shall make explicit in Section 4.3.

3 Adapting effects

> The measure of intelligence is the ability to change.
>
> —often attributed to Albert Einstein

In this section, we motivate the need for a way of reconfiguring access to the ambient ability. We start by discussing the problem of effect pollution and point out how this problem has been solved in the literature so far. Then, we propose the more general solution of *adaptors* which support arbitrary rewiring of effects. We demonstrate the usefulness of the added generality on an example that requires effect duplication and effect swapping. Lastly, we motivate and present *adaptor adjustments*, which are adaptors that allow the precise specification of effect flow at the type-level.

Effect Pollution. We present a small example in order to illustrate a problem that can occur on operator composition. Consider the following two operators:

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

```
receives : {List S -> <Receive S>X -> [Abort]X}
receives []        <receive -> k> = abort!
receives (s :: ss) <receive -> k> = receives ss (k s)
receives _         x              = x
```

The `maybe` handler results in an option type by interpreting an `abort` command as `nothing` and a value `x` as `just x`. The `receives` handler, parameterised by a list of state values, interprets a `receive` command by returning the next value from the list if it exists and by invoking `abort` if the list is empty. Furthermore, it interprets a value by returning it. It is natural to expect to be able to precompose `maybe` with `receives` to obtain an interpretation of the `Receive S` effect as a pure computation. However, this intended operator was not expressible in earlier versions of Frank. Naive composition yields the following operator:

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

As indicated by its type, `bad` handles both `Receive S` and `Abort`. The catch-all pattern `<m>` binds the second argument to a suspended computation `m` of type `{[Receive S, Abort]X}`. As well as handling any `abort` command raised by the `receives` handler, it also handles invocations of `abort` within the argument computation.

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

The second argument to `bad` raises `abort` which is intercepted by `bad`. This is surely not what was intended! Abort effects occurring in the argument computation should be forwarded to the outer context for handling rather than being interpreted by `bad`. The `bad` handler has broken abstraction by leaking its interpretation for an empty list of states and revealing such details in the type of its second argument. We call this phenomenon *effect pollution*. 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 multiple ways to achieve this; alternatives to our approach are discussed in Section 9. The approach we adopt in this paper is a generalisation and adaptation of a construct variously called *inject* (Leijen, 2014, 2018) and *lift* (Biernacki *et al.*, 2018) 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*, supports the arbitrary remapping of effect names and may appear in types and in terms.

Mask. Let us return to our example of precomposing `maybe` with `receives`, which resulted in `bad`. Compare `bad` with the following composed operator `good` which by use of an *adaptor* prohibits the unintentional interception of any external `Abort` effect.

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

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

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

whereas:

```
bad  [1,2] (receive! + 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 renaming de Bruijn representations for bound names.

Remark. The view of 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 can be 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 (Leijen, 2014) or lifting (Biernacki *et al.*, 2018) 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.

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, S' range over patterns that bind the instances of I. An instance pattern consists of a sequence of variables $s\ a_n\ \ldots\ a_1$, where $a_1, \ldots a_n$ bind the first n instances of the interface and s binds the rest (full details are given in Section 4). Examples:

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

The common case of mask omits the map entirely: `I` is syntactic sugar for `I(s a -> s)`. Technically, mask in combination with handlers is sufficient to express general adaptors, but only through a global transformation. For instance, Biernacki *et al.* (2018) express swapping of effects using mask (which they call "lift").

Effect Duplication. We have seen how mask solves the problem of effect pollution. We now give two more examples that demonstrate why the general notion of adaptors gives a powerful tool when it comes to composing effectful components. Consider the following two operators:

```
sqrt:     {Int    -> [Abort]Int}
parseInt: {String -> [Abort]Int}
```

The `sqrt` operator takes an `Int` value and computes its square root, exhibiting the `Abort` effect on negative input. The `parseInt` operator parses a `String` to an `Int` value and may exhibit the `Abort` effect if the string is malformed. There are two sensible ways of composing these operators. One way is to conflate the two effect instances.

```
conflatedComp: {String -> [Abort]Int}
conflatedComp s = sqrt (parseInt s)
```

Thus, only a single exception handler may be used to handle both kinds of failures.

```
maybe (conflatedComp "abc") ⟹ nothing   -- exc. 1
maybe (conflatedComp "-16") ⟹ nothing   -- exc. 2
maybe (conflatedComp "16")  ⟹ just 4
```

Another way of composing the two operators is to distinguish the two effect instances.

```
distinctComp: {String -> [Abort, Abort]Int}
distinctComp s = sqrt (<Abort> (parseInt s))
```

What happens here is that two `Abort` instances are declared, but the inner one is removed from the ambient ability for the argument position of `sqrt`. Since `parseInt` is evaluated in this argument position, effects from `parseInt` are directed to the remaining `Abort` instance, which corresponds to the outer one. Here is an example of how the two different failure scenarios can be handled.

```
maybe (catch (distinctComp "abc") {0}) ⟹ nothing   -- exc. 1
maybe (catch (distinctComp "-16") {0}) ⟹ just 0    -- exc. 2
maybe (catch (distinctComp "16")  {0}) ⟹ just 4
```

We argue that whether to provide `conflatedComp` or `distinctComp` is a design choice, possibly taken by the maintainer of some library. Certainly though, `distinctComp` is more general than `conflatedComp`. It should therefore be easy to obtain `conflatedComp` from `distinctComp` by conflating the two effects, and indeed, it is:

```
conflatedComp': {String -> [Abort]Int}
conflatedComp' s = <Abort(s a -> s a a)> (distinctComp s)
```

In this example, duplication of the `Abort` effect in the ambient ability allows it to be used for both exception scenarios. Although the reconfiguration of the ambient ability is very simple, the adaptor is quite verbose. In future it may be useful to extend Frank to allow the programmer to give names to adaptors by writing something like `Abort(copy)` for `Abort(s a -> s a a)`.

Effect Swapping. Consider the composition of the two exception handlers `maybe` and `catch`.

```
handlerComp: {<Abort, Abort>Int -> Maybe Int}
handlerComp <m> = maybe (catch m! {0})
```

The programmer can now combine these components directly which yields a program that is equivalent to the program above, applied to some string `s`.

```
handlerComp (distinctComp s)
≡ maybe (catch (distinctComp s) {0})
```

Alternatively, adaptors allow the programmer to combine these components with the two `Abort` instances swapped.

```
handlerComp (<Abort(s a b -> s b a)> (distinctComp s))
```

As with effect duplication, we might imagine extending Frank to allow the programmer to name such a common adaptor by writing something like `Abort(swap)` for `Abort(s a b -> s b a)`.

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 includes two instances of the interface, which is usually not desirable.

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

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

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

```
receives [1,2] (inc' (receive! + receive!)) ⟹ 5
```

However, suppose we run `inc'` twice.

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

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

```
incinc'' : {<Receive Int>X -> [Receive Int]X}
incinc'' <m> = inc' (inc' (<Receive(s a b -> s b)> m!))
```

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

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

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

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

The general form for an adjustment is $\Theta|\Xi$ 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,

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

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

Fig. 1. Types.

it would require an unbounded number of effect instances and is therefore not definable in Frank. With adaptors, we can define such a composed operator as follows:

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

4 A Frank formalism

A value is. A computation does.

—Paul Blain Levy (2004)

In this section, we give a formal presentation of the abstract syntax and type system of Frank. 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 `RefState`.) The syntax and typing rules extend those of Lindley *et al.* (2017). 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 (Convent, 2017). We prove a type soundness property (Theorem 7) for the extended system.

4.1 Syntax

We sometime use overline notation ($\overline{M}$) and sometimes explicit indexing ($(M_i)_i$) to denote a sequence $M_1, \ldots, M_n$. The types of Frank (Figure 1) are divided into value types and computation types. Value types are data types $D\ \overline{R}$, suspended computation types $\{C\}$, or type variables X. Computation types are built from zero or more argument types T and one return type G. A computation type

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

has argument types $\langle \Theta_1 \mid \Xi_1 \rangle A_1, \ldots, \langle \Theta_k \mid \Xi_k \rangle A_k$ 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.

An ability Σ represents an infinite map from interface names I to ordered lists of instances $I\ \overline{R}_1, ..., I\ \overline{R}_n$ in which finitely many interface names I map to non-empty lists. In the syntax, the order of instances of the same interface is important but the order of instances of different interfaces is not. An ability may be closed (initiated by $\emptyset$) or open (initiated by an effect variable E).

An extension Ξ represents a finite map from interface names to instances. The extension in an argument type denotes exactly those interfaces which are handled by that argument.

An adaptor Θ represents a finite map from interface names to adaptor components. An adaptor component $I(S \to S')$ specifies a mapping of the ambient ability onto the ability of the argument. It acts only on the ambient ability and not on the extension. An interface pattern S matches a reversed list of interface instances in the ambient ability. s_i is the remaining i interface instances in the ambient ability not bound by explicit pattern variables. We usually omit the subscript i when knowing the number of unbound instances is not significant.

Effect Polymorphism with an Invisible Effect Variable. Consider the type of `map` in Section 2:

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

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:

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

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

$$\langle \Xi\rangle A \equiv \langle \iota \,|\, \Xi\rangle A \qquad\qquad A \equiv \langle \iota \,|\, \iota\rangle A$$

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

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

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. $I(s\ a \to s)$.

Type binders Z may be value binders X or effect binders $[E]$; polytypes may be polymorphic in both value and effect types. 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

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

Fig. 2. Terms.

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.

We have two kinds of environments. Γ binds monomorphic and polymorphic term variables to value types. Ω binds interface pattern variables to interface instances.

Effect Parameters with an Invisible Effect Variable. The body of the definition of a data type or effect interface may use an implicit effect variable. Correspondingly, in this case an implicit parameter is added as the last parameter of the data type definition. We give an example in Section 5.

Abilities and Adjustments. An ability is a collection of interface instances initiated either with the empty ability $\emptyset$ (yielding a *closed* ability) or an effect variable E (yielding an *open* ability). Order is important, as duplicates are permitted, in which case the rightmost interface instance is the one corresponding to the active handler. Closed abilities are not normally required, but they can be used to enforce purity, for instance. In ASCII source code we write $\emptyset$ as `0`.

An adjustment Δ modifies an ability Σ. We introduce a system of inference rules to describe the action of an adjustment $\Delta = \Theta \mid \Xi$ on an ability Σ. The judgements take the general form:

$$\text{Input Contexts} \vdash \text{Transformer} \dashv \text{Output Contexts}$$

where "Transformer" produces one or more output contexts (separated by ";") by acting on one or more input contexts. Such a system of rules is reminiscent of algorithmic presentations of polymorphic type systems (Gundry *et al.*, 2010; Dunfield & Krishnaswami, 2013) which account for and track the knowledge gained—solutions to unification constraints—during the process of typechecking. We utilise the same machinery to account for information increase with respect to the effects offered by the ambient ability: starting from effects globally available to those locally permitted.

Figure 3 presents the judgement for computing the action of an adjustment on an ability along with five further auxiliary judgements:

- $\Sigma \vdash \Delta \dashv \Sigma'$ states that the action of adjustment Δ on ability Σ yields ability Σ';
- $\Sigma \vdash \Xi \dashv \Sigma'$ states that the action of extension Ξ on ability Σ yields ability Σ';
- $\Sigma \vdash \Theta \dashv \Sigma'$ states that the action of adaptor Θ on ability Σ yields ability Σ';
- $\Sigma \vdash I(S \to S') \dashv \Sigma'$ states that the action of adaptor component $I(S \to S')$ on ability Σ yields ability Σ';

- $\Sigma \vdash S : I \dashv \Sigma'; \Omega$ states that adaptor pattern S at interface I yields residual ability Σ' and binds instance environment Ω; and
- $\Omega \vdash S : I \dashv \Xi$ states that in instance environment Ω adaptor pattern S at interface I yields extension Ξ.

The action of an adjustment on an ability (A-ADJ) is obtained by first applying the adaptor to the ability and then applying the extension to the result.

The action of an extension on an ability (the A-EXT-$\star$ rules) is to add each effect instance to the ability. The action of an adaptor on an ability (the A-ADAPT-$\star$ rules) transforms the input ability by applying its individual adaptor components in sequence. The action of an adaptor component (A-ADAPT-COM) first computes the residual ability and instance environment for the input pattern, then uses the instance environment along with the output pattern to compute an extension, which is finally applied to the residual ability. The pattern matching rules (I-PAT-$\star$) yield a residual ability Σ' and an instance environment Ω by matching on variables in S. The instantiation rules (I-INST-$\star$) yield an extension Ξ by instantiating the patterns in S' using the instance environment Ω.

Whereas abilities may contain duplicate effect instances, both the adaptor and extension components of an adjustment must not. For convenience, we sometimes use functional notation for the action of an adjustment, extension, or adaptor on an ability. For instance, we write: $\Xi(\Sigma)$ for Σ' when $\Sigma \vdash \Xi \dashv \Sigma'$. We define the operation $\Sigma \mathbin{@} I$ to be the extension which contains all instances of I occurring in Σ:

$$(\sigma \mid \iota) \mathbin{@} I = \iota \qquad (\Sigma, I\ \overline{R}) \mathbin{@} I = (\Sigma \mathbin{@} I), I\ \overline{R} \qquad (\Sigma, I'\ \overline{R}) \mathbin{@} I = \Sigma \mathbin{@} I, \quad \text{if } I \neq I'$$

We let $|\Xi|$ denote the length of the extension Ξ. We define an indexing operation on extensions. Given an extension Ξ and $0 \leq i < |\Xi|$, define $\Xi(i)$ as follows:

$$(\Xi, I\ \overline{R})(0) = I\ \overline{R} \qquad (\Xi, I\ \overline{R})(i+1) = \Xi(i)$$

This operation is only defined for $0 \leq i < |\Xi|$ so for brevity, whenever we write $\Xi(i)$ we implicitly assume that i satisfies this inequality.

Terms. Frank follows a bidirectional typing discipline (Pierce & Turner, 2000). Thus, terms (Figure 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 (**let** $f : P = n$ **in** n'), mutual recursion (**letrec** $\overline{f : P = e}$ **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 ! for the 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 $\overline{p}$ (which are in turn bound) 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.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Action of an adjustment on an ability and auxiliary judgements.

Example. To illustrate how source programs may be represented as abstract syntax, we give the abstract syntax for an example involving the `map`, `state`, and `index` operators from Section 2.

letrec map :
  $\forall \varepsilon\ X\ Y.\{\langle \iota \mid \iota \rangle \{\langle \iota \mid \iota \rangle X \to [\varepsilon \mid \iota] Y\} \to \langle \iota \mid \iota \rangle (\mathsf{List}\ X) \to [\varepsilon \mid \iota](\mathsf{List}\ Y)\} =$
    $f\ \mathsf{nil} \mapsto \mathsf{nil}$
    $f\ (\mathsf{cons}\ x\ xs) \mapsto \mathsf{cons}\ (f\ x)\ (\mathsf{map}\ \varepsilon\ X\ Y\ f\ xs)$ **in**
letrec state : $\forall \varepsilon\ X\ S.\{\langle \iota \mid \iota \rangle X \to \langle \iota \mid \iota, \mathsf{State}\ S \rangle X \to [\varepsilon \mid \iota] X\}$
    $= s\ x \mapsto x$
      $s\ \langle \mathsf{get} \mapsto k \rangle \mapsto \mathsf{state}\ \varepsilon\ X\ S\ s\ (k\ s)$
      $s\ \langle \mathsf{set}\ s' \mapsto k \rangle \mapsto \mathsf{state}\ \varepsilon\ X\ S\ s'\ (k\ \mathsf{unit})$ **in**
let index : $\forall \varepsilon\ X.\{\langle \iota \mid \iota \rangle \mathsf{List}\ X \to [\varepsilon \mid \iota] \mathsf{List}\ (\mathsf{Pair}\ \mathsf{Nat}\ X)\} =$
    $\{xs \mapsto \mathsf{state}\ \varepsilon\ X\ \mathsf{Nat}\ \mathsf{zero}\ (\mathsf{map}\ \varepsilon\ X\ (\mathsf{Pair}\ \mathsf{Nat}\ X)\ \{x \mapsto \mathsf{pair}\ \mathsf{next!}\ x\}\ xs)\}$ **in**
index $[\emptyset \mid \iota]$ String "abc"

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

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

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

$$\textsc{WF-Data}\ \frac{(\Phi \vdash R)_i}{\Phi \vdash D\ \overline{R}} \qquad \textsc{WF-Thunk}\ \frac{\Phi \vdash C}{\Phi \vdash \{C\}} \qquad \textsc{WF-Comp}\ \frac{(\Phi \vdash T)_i \qquad \Phi \vdash G}{\Phi \vdash \overline{T} \to G}$$

$$\textsc{WF-Arg}\ \frac{\Phi \vdash \Delta \qquad \Phi \vdash A}{\Phi \vdash \langle \Delta \rangle A} \qquad \textsc{WF-Ret}\ \frac{\Phi \vdash \Sigma \qquad \Phi \vdash A}{\Phi \vdash [\Sigma] A}$$

$$\textsc{WF-Ability}\ \frac{\Phi \vdash \Sigma}{\Phi \vdash [\Sigma]} \qquad \textsc{WF-Pure}\ \frac{}{\Phi \vdash \emptyset} \qquad \textsc{WF-Id}\ \frac{}{\Phi \vdash \iota} \qquad \textsc{WF-Ext}\ \frac{\Phi \vdash \Xi \qquad (\Phi \vdash R)_i}{\Phi \vdash \Xi, I\ \overline{R}} \qquad \textsc{WF-Adapt}\ \frac{\Phi \vdash \Theta}{\Phi \vdash \Theta, I\ (S \to S')}$$

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

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

Fig. 4. Well-formedness rules.

The map function and state handler are recursive, so are defined using **letrec**, whereas the index function is not recursive so is defined with **let**. The type signatures are adorned with explicit universal quantifiers and braces to denote that they each define suspended computations. Pattern matching by equations is represented by explicit pattern matching in the standard way. Each wildcard pattern is represented with a fresh variable.

Frank Design. Inspiration for the Frank is drawn from Levy's CBPV calculus (Levy, 2004). Indeed, as in CBPV, we separate value and computation types. However, the similarities really end there. Frank has no equivalent to the CBPV "producer" (F) types, and more significantly, Frank does not distinguish *terms* by whether they are a computation or a value. Indeed, CBPV's eponymous rule for always applying a function to a *value* does not carry over to Frank which instead generalises functional abstraction to incorporate effect handling *in situ*. Our design decision is motivated by a desire to reduce boilerplate in source programs and contrasts with other languages incorporating effect handling (Pretnar, 2009; Bauer & Pretnar, 2014; Ahman, 2017), where a separate handling construct is introduced. Hence, Frank combines the features of CBPV, call-by-value (a la ML), and effect typing to smoothly integrate well-typed effectful code within the functional paradigm.

4.2 Typing rules

Before defining the typing judgements for Frank, we first define the well-formedness judgement in Figure 4: $\Phi \vdash \mathscr{X}$ states that $\mathscr{X}$ is well-formed in kind environment Φ. (The meta variable $\mathscr{X}$ ranges over the various syntactic categories defined in Figure 1 as well as existential type environments $\exists \Psi.\Gamma$, which will be used for typing handlers

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

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

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

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

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

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

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

$$\textsc{T-Adapt}\ \frac{\Sigma \vdash \Theta \dashv \Sigma' \qquad \Phi;\Gamma\,[\Sigma']\vdash n : A}{\Phi;\Gamma\,[\Sigma]\vdash \langle\Theta\rangle\ n : A}$$

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

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

Fig. 5. Term typing rules.

for polymorphic commands (the T-Comp rule in Figure 5 and the P-Command rule in Figure 6).) The well-formedness rules are routine, following the structure of $\mathscr{X}$. The well-formedness judgement is to the usual intuitionistic structural rules (weakening, contraction, and exchange).

The term typing rules for Frank are given in Figure 5. Each judgement is defined with respect to kind environment Φ and is implicitly subject to the condition that each of its subcomponents must be well-formed with respect to Φ:

- $\Phi;\Gamma\,[\Sigma]\vdash m \Rightarrow A$ states that in type environment Γ with ambient ability Σ, we can infer that use m has type A;
- $\Phi;\Gamma\,[\Sigma]\vdash n : A$ states that in type environment Γ with ambient ability Σ, construction n has type A;
- $\Phi;\Gamma \vdash e : C$ states that in type environment Γ, computation e has type C;

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

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

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

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

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

Fig. 6. Pattern matching typing rules.

- $\Phi \vdash r : T \dashv [\Sigma]\ \exists \Psi.\Gamma$ states that computation pattern r of argument type T with ambient ability Σ binds existentially quantified type environment $\exists \Psi.\Gamma$;
- $\Phi \vdash p : A \dashv \Gamma$ states that value pattern p of type A binds type environment Γ.

The first three judgements are subject to the usual intuitionistic structural rules (weakening, contraction, and exchange). 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 modified by 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.

The T-Switch rule allows us to treat a use as a construction. The checking rules for data types (T-Data), suspended computations (T-Thunk), polymorphic let (T-Let), mutual recursion (T-LetRec), and adaptors (T-Adapt) recursively check the sub-terms. 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 $\overline{r} \mapsto n$ (T-Comp), where $\overline{r}$ is a sequence of computation patterns whose variables are bound in n. The side condition in the T-Comp rule requires that the patterns in the clauses cover all possible values inhabiting the argument types. While we do not discuss coverage checking any further here, Lindley *et al.* (2017) have presented an elaboration algorithm for Frank which checks coverage.

The typing rules for pattern matching are presented in Figure 6. Value patterns match variables (P-Var) and data type constructor applications (P-Data). Value patterns can be typed as computation patterns (P-Value). Catch-all patterns suspend the computation

(P-CATCHALL). For both value patterns and catch-all patterns, we apply the adjustment to the ambient ability and we omit the existential in the conclusion as Ψ is empty. 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. The existential quantifier ensures that the type variables $\overline{Z}$ cannot escape.

Instantiating ε. In an earlier version of Frank, the POLYVAR rule was restricted to always instantiate the implicit effect variable ε with the ambient ability Σ. Correspondingly, data types were restricted to being parameterised by at most one effect variable, namely ε. The language resulting from these restrictions has the pleasant property that effect variables need never be written at all. However, we now feel that the restrictions are artificial, and having multiple effect variables may be useful. Given that the APP rule already checks that abilities match up exactly where needed, relaxing the POLYVAR rule does no harm, and now we can support data types parameterised by multiple effect variables.

4.3 Operational semantics

Definition 1 (Simultaneous Substitution)**.** *For type environments Γ and Γ', let $\Phi \vdash \theta : \Gamma \vDash \Gamma'$ denote the simultaneous substitution θ of variables in environment Γ by use values which are well-typed in environment Γ'. More formally, we define the judgement $\Phi\ [\Sigma]\vdash\ \theta : \Gamma \vDash \Gamma'$ via the following rule:*

$$\frac{(\text{for all}(x : A) \in \Gamma \mid \Phi;\ \Gamma'\ [\Sigma]\vdash\ \theta(x) \Rightarrow A) \qquad (\text{for all}(f : \forall \overline{Z}.A) \in \Gamma, \overline{R} \mid \Phi \vdash A[\overline{R}/\overline{Z}] \implies \Phi;\ \Gamma'\ [\Sigma]\vdash\ \theta(f\ \overline{R}) \Rightarrow A[\overline{R}/\overline{Z}])}{\Phi\ [\Sigma]\vdash\ \theta : \Gamma \vDash \Gamma'}$$

Since the type of a use value is independent of the ambient ability, we write simultaneous substitutions as $\Phi \vdash \theta : \Gamma \vDash \Gamma'$ for short. We write $[u/x]$ for the unary special case substituting a use value u for a variable x at some type A, corresponding to $\Gamma \vDash \Gamma, x : A$ for any Γ. We write $[\overline{u}/\overline{x}]$ for the simultaneous substitution, substituting each u_i for each x_i. Abusing notation slightly, we write $[\uparrow(w : \forall \overline{Z}.A)/f]$ for the substitution of $\uparrow(w : A[\overline{R}/\overline{Z}])$ for $f\ \overline{R}$ with a corresponding generalisation for simultaneous substitution.

Our typing rules are closed under standard weakening so we may extend the codomain of any simultaneous substitution with a free (poly) variable.

Lemma 2. *Given type environments Γ and Γ', and simultaneous substitution $\Phi \vdash \theta : \Gamma \vDash \Gamma'$, we have:*

- *for all $(f : P) \notin \Gamma'$, the judgement $\Phi \vdash \theta : \Gamma \vDash \Gamma', f : P$ holds;*
- *for all $(x : A) \notin \Gamma'$, the judgement $\Phi \vdash \theta : \Gamma \vDash \Gamma', x : A$ holds.*

Lemma 3 (Substitution Preserves Typing)**.** *Given Γ, Γ' and $\Phi \vdash \theta : \Gamma' \vDash \Gamma$:*

- *If $\Phi;\ \Gamma\ [\Sigma]\vdash\ m \Rightarrow A$, then $\Phi;\ \Gamma'\ [\Sigma]\vdash\ \theta(m) \Rightarrow A$;*

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

Fig. 7. Runtime syntax.

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

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

Fig. 8. Frozen commands.

- *If* $\Phi; \Gamma\ [\Sigma] \vdash n : A$, *then* $\Phi; \Gamma'\ [\Sigma] \vdash \theta(n) : A$;
- *If* $\Phi; \Gamma \vdash e : C$, *then* $\Phi; \Gamma' \vdash \theta(e) : C$.

Runtime Syntax. The operational semantics relies on the runtime syntax defined in Figure 7. Both uses and constructions are augmented with a special form $\lceil \mathscr{E}[c\ \overline{R}\ \overline{w}] \rceil$ of *frozen commands* described below. 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 commands. 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, evaluation contexts also ramify as such, leading to four distinct classes; for each possible combination of hole and body (see the R-LIFT-* rules in Figure 9). However, we do not explicitly distinguish the four kinds of evaluation contexts 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 $\lceil \mathscr{E}[c\ \overline{R}\ \overline{w}] \rceil$ is a command application $c\ \overline{R}\ \overline{w}$ plugged inside an evaluation context $\mathscr{E}$ that does not handle c. Frozen commands only occur at runtime (they cannot be written in programs). The typing rules for frozen commands are given in Figure 8. Evaluation contexts can either be uses or constructions; hence, we need two typing rules. The side conditions depend on an auxiliary predicate $\mathscr{E}$ handles c, which is true if c is handled by $\mathscr{E}$.

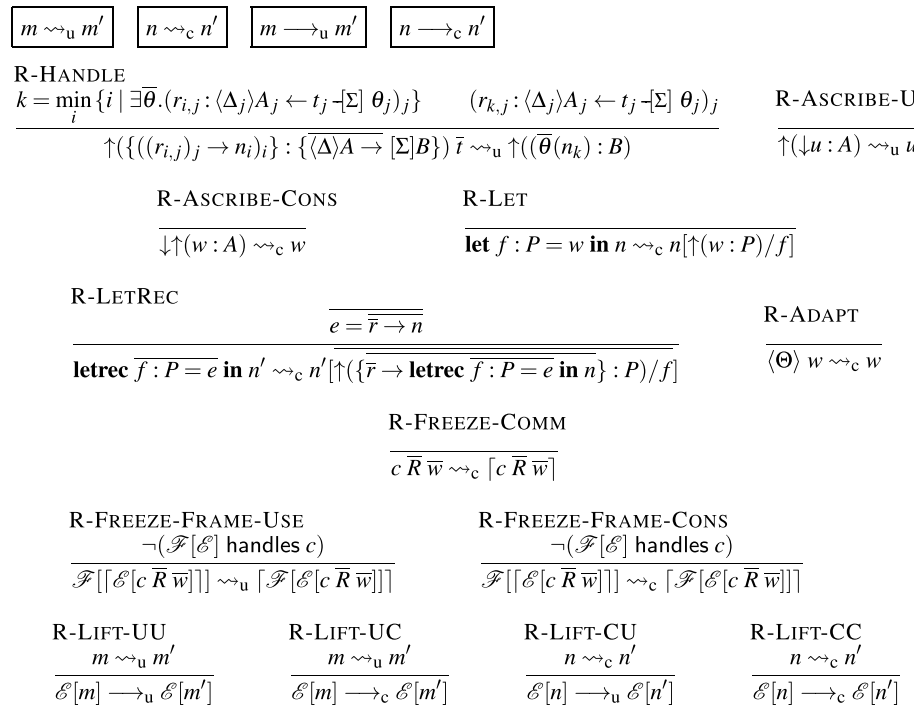

Fig. 9. Operational semantics.

Evaluation Context Adaptor Functions. In order to determine whether an evaluation context handles a given command, we first compute an adaptor function which describes how the evaluation context transforms effects. In order to account for extensions, we write $\bot$ (a distinguished pattern variable) to denote a concrete instantiation of an interface. For instance

$$s \to s\ \bot$$

is an adaptor function that denotes the handling of an interface, and

$$s\ a \to s\ \bot$$

is an adaptor function that denotes erasing one copy of an interface from the ambient ability and then handling another copy, and is obtained as the composition:

$$(s\ a \to s); (s \to s\ \bot)$$

Composition is defined by substitution:

$$(s\ \overline{a_1} \to s\ \overline{a_2}); (s\ \overline{b_1} \to s\ \overline{b_2}) = s\ \overline{a_1} \to s\ \overline{b_2}[\overline{a_2}/\overline{b_1}]$$

such that $|\overline{a_2}| = |\overline{b_1}|$. We can always ensure that the arities match up using the equation:

$$(s\ \overline{a} \to s\ \overline{v}) = (s\ a\ \overline{a} \to s\ a\ \overline{v})$$

for some fresh a.

The adaptor function of an evaluation context is defined by composing together the adaptor functions of its constituent evaluation frames.

$$\mathsf{af}(c, [\,]) = s \to s$$
$$\mathsf{af}(c, \mathscr{F}[\mathscr{E}]) = \mathsf{af}(c, \mathscr{F}); \mathsf{af}(c, \mathscr{E})$$

The adaptor function for an evaluation frame is the identity $(s \to s)$ for all frames except argument frames and adaptor frames for which it is defined as follows:

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

The adaptor function $\mathsf{af}(c, \Xi)$ for a command c in an extension Ξ is as follows:

$$\mathsf{af}(c, \iota) = s \to s$$
$$\mathsf{af}(c, (\Xi, I\ \overline{R})) = \begin{cases} s \to s\ \bot, & \text{if } c \in I \\ \mathsf{af}(c, \Xi), & \text{otherwise} \end{cases}$$

The adaptor function $\mathsf{af}(c, \Theta)$ for a command c in an adaptor Θ is as follows:

$$\mathsf{af}(c, \iota) = s \to s$$
$$\mathsf{af}(c, (\Theta, I(S \to S'))) = \begin{cases} S \to S', & \text{if } c \in I \\ \mathsf{af}(c, \Theta), & \text{otherwise} \end{cases}$$

The predicate $\mathscr{E}$ handles c holds iff $\mathsf{af}(c, \mathscr{E})$ has the form $S \to S'\ \bot$.

We also define another predicate $\mathscr{E}$ poisedfor c which captures the intuition that a c handler placed around $\mathscr{E}[c\ \overline{u}]$ will handle c. The predicate $\mathscr{E}$ poisedfor c holds iff $\mathsf{af}(c, \mathscr{E})$ is equivalent to an adaptor function of the form $S\ a \to S'\ a$ or $s \to s$. So, in particular, the empty evaluation context is always waiting for the nearest dynamically enclosing handler to handle any command.

The adaptor function computes a transformation on the ambient ability. That is, the direction of information flow is inwards, computing the local ability by transformation on the effects offered globally. Of course, we can reverse the direction of information flow and compute the ambient ability by retracing our steps through the evaluation frames. This second viewpoint captures the notion of effect handling and forwarding which we formalise via a *rewiring* operation on the output of adaptor functions. Given a syntactic phrase class $\mathscr{P}$ such that $\mathsf{af}(c, \mathscr{P}) = (s\ \overline{a} \to s\ \overline{b})$ and natural number i, we define the operation $\mathsf{af}(c, \mathscr{P})(i)$ or, equivalently $(s\ \overline{a} \to s\ \overline{b})(i)$, as follows:

$$(s\ \overline{a} \to s\ \overline{b})(i) = \begin{cases} j, & \text{if } b_i = a_j \\ \bot, & \text{if } b_i = \bot \\ i + |\overline{a}| - |\overline{b}| & \text{if } i \geq |\overline{b}| \end{cases}$$

Recall, the list of patterns are snoc lists and indexing is right-to-left. So, for example, index 0 corresponds to the rightmost instance in the list. The rewiring operation is a crucial component for proving the Frank type system is sound. Indeed, we must establish a connection between rewiring and the static transformations on abilities performed by the type system. Together, they encompass the two viewpoints outlined above. The statics describe how the ambient ability offered at the top-level is transformed and subsequently offered locally. The dynamics (rewiring) describe how to satisfy a local request by forwarding the request upwards through evaluation frames, arriving at the top-level if and only if the request can be satisfied by the ability. Later, we define a relationship between normal forms and abilities (Definition 6) which formally characterises this connection between the static and dynamic viewpoints.

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

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

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

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

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

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

$$\textsc{B-Var}\ \frac{}{x : A \leftarrow w \dashv [\uparrow(w : A)/x]}$$

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

Fig. 10. Pattern binding.

The semantics is given in Figure 9. We define four reduction relations: top-level use reduction ($m \rightsquigarrow_{\mathrm{u}} m'$), top-level construction reduction ($n \rightsquigarrow_{\mathrm{c}} n'$), full use reduction ($m \longrightarrow_{\mathrm{u}} m'$), and full construction reduction ($n \longrightarrow_{\mathrm{c}} n'$).

The R-Handle (Figure 9) rule depends on two auxiliary judgements (Figure 10). The judgement $r : T \leftarrow t \dashv[\Sigma]\ \theta$ states that computation pattern r of argument type T at ability Σ matches normal form t yielding substitution θ. The judgement $p : A \leftarrow w \dashv[\Sigma]\ \theta$ 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-$\star$ rules extend 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.

Lemma 4 (Substitution from Pattern Binding)**.**

- *If* $r : \langle\Delta\rangle A \leftarrow t \dashv[\Sigma]\ \theta$ *and* $\Phi \vdash r : \langle\Delta\rangle A \dashv[\Sigma]\ \Gamma$ *and* $\Sigma \vdash \Delta \dashv \Sigma'$ *and* $\Phi; \Gamma'\ [\Sigma']\vdash\ t : A$*, then* $\Phi \vdash \theta : \Gamma \vDash \Gamma'$.
- *If* $p : A \leftarrow w \dashv \theta$ *and* $\Phi \vdash p : A \dashv \Gamma$ *and* $\Phi; \Gamma'\ [\Sigma]\vdash\ w : A$*, then* $\Phi \vdash \theta : \Gamma \vDash \Gamma'$.

Proof. By simultaneous induction on derivations $r : T \leftarrow t \dashv[\Sigma]\ \theta$ and $p : A \leftarrow w \dashv \theta$, and inversion of the typing assumptions. □

Reduction preserves typing.

Theorem 5 (Subject Reduction)**.**

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

Proof. By simultaneous induction on the transitions $m \rightsquigarrow_{\mathrm{u}} m'$ and $n \rightsquigarrow_{\mathrm{c}} n'$, and inversion on the typing assumptions. See the Appendix for details. □

Programs are constructions. If a program stops reducing, then it must be a normal form, that is, either a value or a frozen command (and the latter only if the ambient ability is non-empty). We formally specify normal forms in Definition 6, complete with necessary side conditions that marry the static semantics enforced by the Frank type system and the dynamic semantics enforced by the operational semantics. Crucially, the definition connects the action of a construction on the ambient ability, which is computed statically by the typing rules, to the dynamic rewiring performed by the adaptor function. The side conditions tell us that any command which is forwarded all the way to the top-level was in fact offered by the ambient ability.

Definition 6. *We say that normal form t respects Σ if it is either a value w or a frozen command $\lceil \mathscr{E}[c\ \overline{R}\ \overline{w}] \rceil$ such that $c \in I$ and $\mathsf{af}(c, \mathscr{E})(0) = i < |\Sigma@I|$ and $c\ \overline{R} \in (\Sigma@I)(i)$.*

Theorem 7 (Type Soundness)**.**

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

In particular, if $\Sigma = \emptyset$, then either the term is a value w or the term can reduce by one step.

Proof. By simultaneous induction on the typing derivations $\cdot; \cdot\ [\Sigma]\vdash\ m \Rightarrow A$ and $\cdot; \cdot\ [\Sigma]\vdash\ n : A$ using subject reduction (Theorem 5). Additionally, the proof relies on a number of auxiliary lemmas connecting the static typing rules for the action of an adjustment on an ability (Figure 3), and the rewiring operation $\mathsf{af}(-, -)(-)$ on adaptor functions. We present the full details of the proof in the Appendix. □

5 Computations as data

So far, our example data types have been entirely first order, but our type system admits data types which abstract over abilities exactly to facilitate the storage of suspended computations in a helpfully parameterised way. When might we want to do that? Let us develop an example, motivated by Shivers and Turon's treatment of *modular rollback* in parsing (Shivers & Turon, 2011).

Consider a high-level interface to an input stream of characters with one-step lookahead. A parser may `peek` at the next input character without removing it, and `accept` that character once its role is clear.

```
interface LookAhead = peek : Char | accept : Unit
```

We might seek to implement `LookAhead` on top of regular `Console` input, specified thus:

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

where an input of `'\b'` indicates that the backspace key has been struck. The appropriate behaviour on receipt of backspace is to unwind the parsing process to the point where the previous character was first used, then await an alternative character. To achieve that unwinding, we need to keep a *log*, documenting what the parser was doing when the console actions happened.

```
data Log X = start  {X}
           | inched (Log X) {Char -> X}
           | ouched (Log X)
```

Note that although `Log` is not explicitly parameterised by an ability, it must be implicitly parameterised as it stores implicitly effect-polymorphic suspended computations. The above definition is shorthand for the following:

```
data Log X [ε] = start {[ε]X}}
               | inched (Log X [ε]) {Char -> [ε]X}
               | ouched (Log X [ε])
```

As discussed in Section 4, the general rule is that if the body of a data type (or interface) definition includes the implicit effect variable, then the last parameter of the definition is also the implicit effect variable. Initially (`start`) a log contains a parser computation. When a character is input (`inched`) the log is augmented with the continuation of the parser, which depends on the character read, allowing the continuation to be replayed if the input character changes. When a character is output (`ouched`) there is no need to store the continuation as the return type of an output is `Unit` and hence cannot affect the behaviour of the parser.

Modular rollback can now be implemented as a handler informed by a log and a one character buffer.

```
data Buffer = empty | hold Char
```

The parser process being handled should also be free to reject its input by aborting, at which point the handler should reject the character which caused the rejection.

```
input : {Log X [LookAhead, Abort] -> Buffer ->
           <Console|LookAhead, Abort>X -> [Console]X}
input _ _         x              = x
input l (hold c) <peek -> k>    = input l (hold c) (k c)
input l (hold c) <accept -> k> = ouch c;
                                   input (ouched l) empty
                                         (k unit)
input l empty    <accept -> k> = input l empty (k unit)
input l empty    <peek -> k>   =
  case inch!
  { '\b' -> rollback l
  | c    -> input (inched l k) (hold c) (k c) }
input l _        <abort -> k>  = rollback l
```

Note the use of an adaptor adjustment in the third argument to remove the `Console` effect from the ambient ability. In doing so, we can ascribe a more precise ability to the `Log` type, prohibiting the storage of continuations which perform `Console` commands. Correspondingly, it is clear that the parser's continuations may be stored, and under which conditions those stored continuations can be invoked, when we `rollback`.

```
rollback : {Log X [LookAhead, Abort] -> [Console]X}
rollback (start p)    = parse p
rollback (ouched l)   = map ouch "\b \b"; rollback l
rollback (inched l k) = input l empty (k peek!)

parse : {{[LookAhead, Abort]X} -> [Console]X
parse p = input (start p) empty p!}
```

To undo an `ouch`, we send a backspace, a blank, and another backspace, erasing the character. To undo the `inch` caused by a "first `peek`", we empty the buffer and reinvoke the old continuation after a new `peek`.

Here is a basic parser that accepts a sequence of zeros terminated by a space, returning the total length of the sequence on success, and aborting on any other input.

```
zeros : {Int -> [LookAhead, Abort]Int}
zeros n = on peek! { '0' -> accept!; zeros (n+1)
                   | ' ' -> accept!; n
                   | c   -> abort!}
```

In order to implement actual console input and output, the `Console` interface is handled specially at the top-level using a built-in handler that interprets `inch` and `ouch` as actual character input and output. The entry point for a Frank program is a nullary `main` operator.

```
main : {[Console]Int}
main! = parse (zeros 0)
```

The ability of `main` is the external ability of the whole program. We can use it to configure the runtime to the execution context: is it a terminal? is it a phone? is it a browser? What will the user *let* us do? Currently, Frank supports a limited range of built-in top-level interfaces, but one can imagine adding many more and in particular connecting them to external APIs.

We have revisited an example presented by Lindley *et al.* (2017) but have been able to give a more constrained type to the parser process, preventing it from accessing `Console` commands by using an adaptor in the signature for `input` to mask this effect. No changes to the implementation were necessary.

6 Concurrency in Frank

In this section, we implement concurrent actors with mailboxes in Frank. This application serves also as an example for the usefulness of adaptors, as a comparison to an earlier implementation of this system shows (Convent, 2017). We present the actor system 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 structure (Huet, 1997).

```
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 manipulate the queue by themselves, 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! = on 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 adaptor, 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> = on (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 (Convent, 2017):

```
{<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 a -> s a a)> (act (<RefState> chain'!))
```

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

7 Implementation

The third author has been plotting Frank since at least 2007 (McBride, 2007). In 2012, he implemented a prototype for a previous version of Frank (McBride, 2012). Since then the design has evolved. A significant change is the introduction of operators that handle multiple computations simultaneously. More importantly, a number of flaws in the original design have been ironed out as a result of formalising the type system and semantics.

We have implemented a prototype of Frank (Convent *et al.*, 2020) in Haskell following the design described in the current paper. In order to rapidly build a prototype, we consciously decided to take advantage of a number of existing technologies. The current prototype takes advantage of the indentation sensitive parsing framework of Adams & Ağacan (2014), the "type-inference-in-context" technique of Gundry *et al.* (2010), and the existing implementation of shonky (McBride, 2016), which we have modified slightly to support features like adaptors and references.

Much like Haskell, in order to aid readability, the concrete syntax of Frank is indentation sensitive (though we do not explicitly spell out the details in the paper). In order to implement indentation sensitivity, Adams & Ağacan (2014) introduce an extension to parsing frameworks based on parsing expression grammars. Such grammars provide a formal basis for the Parsec (Leijen & Martini, 2015) and Trifecta (Kmett, 2015) parser combinator Haskell libraries. In contrast to the ad hoc methods typically employed by many indentation sensitive languages (including Haskell and Idris Brady, 2013a), Adams & Ağacan's extension has a formal semantics. Frank's parser is written using Trifecta with the indentation sensitive extension, which greatly simplifies the handling of indentation by separating it as much as possible from the parsing process.

For bidirectional typechecking, our prototype uses Gundry *et al.*'s "type-inference-in-context" technique (Gundry *et al.*, 2010) for implementing type inference and unification

(Gundry's thesis Gundry, 2013 contains a more detailed and up-to-date account). The key insight is to keep track in the context not just the term variables but also the unification variables. The context enforces an invariant that later bindings may only depend on earlier bindings. The technique has been shown to scale to the dependently typed setting (Gundry, 2013).

For convenience, the implementation of Frank provides *interface aliases* which name the composition of several interfaces. For instance, we can compose interfaces for sending and receiving to obtain a state interface.

```
interface Send X    = send : X -> Unit
interface Receive X = receive : X
interface State X   = [Send X, Receive X]
```

The back end of Frank is implemented by first translating to shonky (McBride, 2016), which amounts to an untyped version of Frank. The shonky implementation executes code directly through an abstract machine much like that of Hillerström *et al.* (2020). The evaluation contexts in the formalism correspond to stack frames in the abstract machine.

8 Adaptor 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 Section 4, the action of an adjustment on an ability first applies the adaptor and then applies the extension.

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

If we were to switch the order

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

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 a b -> s b a)|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 a -> s (Int) a)` an adjustment that handles the second instance of `Reader`.

Adaptive Abilities. A non-uniformity 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:

$$\Sigma ::= \sigma \mid \Delta$$

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 a b -> s b)` to the ability `0|Reader Int`, then `b` is bound to the singleton list containing the one instance of `Reader Int` and `a` 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 typecheck. 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 a -> s)` and `Reader(s a b -> s b)`.

9 Related work

We have discussed much of the related work throughout the paper. Here, we briefly mention some other related work.

Algebraic Theory of Effects and Handlers. Effect handlers were originally built on Plotkin and Power's algebraic theory of effects (Plotkin & Power, 2001a,b, 2002, 2003). In Plotkin and Power's setting, algebraic effects are accompanied by an equational theory governing the behaviour of the operations. In all of the implementations we have discussed (Eff, Helium, Koka, Links), and in Frank, the equational theory is taken to be the free theory, in which there are no equations. Moreover, none of these implementations guarantees that effect handlers are algebraic.

Efficient Effect Handler Implementations. A natural implementation for handlers is to use *free monads* (Kammar *et al.*, 2013). Swierstra (2008) illustrates how to write effectful programs with free monads in Haskell, taking advantage of type classes to provide a certain amount of modularity. However, using free monads directly can be quite inefficient (Kammar *et al.*, 2013).

Wu & Schrijvers (2015) show how to obtain a particularly efficient implementation of deep handlers taking advantage of fusion. Their work explains how Kammar *et al.* (2013) implement efficient handler code in Haskell. Kiselyov & Ishii (2015) optimise their shallow effect handlers implementation, which is based on free monads, by taking advantage of an efficient representation of sequences of monadic operations (van der Ploeg & Kiselyov, 2014). The experimental multicore extension to OCaml (Dolan *et al.*, 2015) extends OCaml with effect handlers motivated by a desire to abstract over scheduling strategies. It does not include an effect system. It does provide an efficient implementation by optimising for the common case in which continuations are invoked at most once (the typical case for a scheduler). The implementation uses the stack to represent continuations and as the continuation is used at most once there is no need to copy the stack. Koka (Leijen, 2017) takes advantage of a selective CPS translation to improve the efficiency of generated JavaScript code.

Layered Monads and Monadic Reflection. Filinski's work on monadic reflection (Filinski, 2010) and layered monads (Filinski, 1999) is closely related to effect handlers. Monadic reflection supports a similar style of composing effects. The key difference is that monadic reflection interprets monadic computations in terms of other monadic computations, rather than abstracting over and interpreting operations.

Swamy *et al.* (2011) add support for monads in ML, providing direct-style effectful programming for a strict language. Unlike Frank, their system is based on monad transformers rather than effect handlers.

Schrijvers *et al.* (2019) compare the expressiveness of effect handlers and monad transformers in the context of Haskell. Forster *et al.* (2017, 2019) compare effect handlers with monadic reflection and delimited control in a more abstract untyped setting; Piróg *et al.* (2019) extend this approach in order to compare effect handlers with delimited control in the presence of types, making essential use of polymorphism.

Variations and Applications. Lindley (2014) investigates an adaptation of effect handlers to more restrictive forms of computation based on idioms (McBride & Paterson, 2008) and arrows (Hughes, 2004). Wu *et al.* (2014) study scoped effect handlers. They attempt to tackle the problem of how to modularly weave an effect handler through a computation whose commands may themselves be parameterised by other computations. Kiselyov & Ishii (2015) provide solutions to particular instances of this problem. Schrijvers *et al.* (2014) apply effect handlers to logic programming.

Effect Instances. Brady (2013b) provides a lightweight syntax for statically supporting multiple instances of the same effect. Early versions of the Eff programming language (Bauer & Pretnar, 2014, 2015) 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 (Hillerström & Lindley, 2016; Hillerström *et al.*, 2020), a fresh effect construct would suffice, as there is no need for mask anyway.

Effect Instances and Lexically Scoped Effect Handlers. Biernacki *et al.* (2020) propose a novel form of effect instances using lexically scoped effect handlers. Lexically scoped effect handlers are not quite as expressive as the usual dynamically scoped effect handlers, but it remains to be seen whether the restricted expressive power is a significant impediment in practice. In his PhD thesis, Bram Geron studies the foundations for algebraic lexically scoped effect handlers through a construct he calls *defined algebraic operations* (Geron, 2019).

Inject, Lift, and Mask. Inject, lift, and mask are different names for the same construct. Leijen's Koka language (Leijen, 2014) 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 (Leijen, 2014) as a way to address a particular instance of the effect pollution problem for hard-coded exceptions. Koka now supports effect handlers (Leijen, 2017) and a generalised inject for arbitrary effects (Leijen, 2018). Biernacki *et al.* (2018) 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.* (2019) 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 (Dolan *et al.*, 2015) supports effect handlers without effect types. The OCaml module system provides a mechanism for simulating effect instances. Biernacki *et al.*'s programming language Helium (Biernacki *et al.*, 2019) also incorporates a module system, which is elaborated into λ^{HEL} using existential types.

Negative Adjustments. In an earlier version of this paper (Lindley *et al.*, 2017) 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 & Myers (2019) 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. (2018), 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.

10 Future work

We have further progress to make on many fronts, theoretical and practical.

Multihandlers in other Calculi. Multihandlers appear to share some features of other expressive programming abstractions such as multiple dispatch and join patterns (Fournet & Gonthier, 1996), and it would be interesting to better understand how multihandlers relate to these abstractions.

Verbs versus Nouns. Our rigid choice that names stand for values means that nullary operators need `!` to be invoked. They tend to be much more frequently found in the doing than the being, so it might be prettier to let a name like `jump` stand for the "intransitive verb", and write `{jump}` for the "noun". Similarly, there is considerable scope for supporting conveniences such as giving functional computations by partial application whenever it is unambiguous.

Data as Computations. Frank currently provides both interfaces and data types. However, given an empty type 0, we can simulate lazy coinductive data types using interfaces (and data constructors using commands). Data constructors can be seen as exceptions, that is, commands that do not return a value. For instance, we can encode lazy possibly infinite lists as follows:

```
interface ListI X = nil  : Zero
                  | cons : X -> {[ListI X]Zero} -> Zero
```

The type of lists with elements of type `X` is `{[ListI X]Zero}`. We can then simulate `map` as follows:

```
mapI : {{X -> Y} -> <ListI X>Zero -> [ListI Y]Zero}
mapI f <nil        -> _> = nil!
mapI f <cons x xs -> _> = cons (f x) (mapI f xs!)
```

Note that the pattern matching clauses are complete because the return type is uninhabited.

Given that computations denote free monads (i.e. trees) and data types also denote trees, it is hardly surprising that there is a correspondence here. Indeed Atkey's effect handlers account of typechecking and elaboration (Atkey, 2015) makes effective use of this correspondence. We would like to study abstractions for more seamlessly translating back and forth between computations and data.

Failure and Choice in Pattern Matching. We could extend Frank to realise incomplete or ambiguous patterns as effects. Pattern matching can fail (if the patterns are incomplete) or it can succeed in multiple ways (if the patterns are redundant). Thus, in general pattern, matching yields a searchable solution space. We can mediate failure and choice as effects,

separating what it is to *be* a solution from the strategy used to *find* one. Concretely, we envisage the programmer writing custom failure and choice handlers for navigating the search space. Wu *et al.* (2014) have shown the modularity and flexibility of effect handlers in managing backtracking computations: the design challenge is to deploy that power in the pattern language as well as in the expression language.

Controlled Snooping. We might consider allowing handlers to trap some or even all commands generically, just as long as their argument types make this possibility clear. Secret interception of commands remains anathema.

Indexed Interfaces. Often, an interaction with the environment has some sort of state, affecting which commands are appropriate, e.g. reading from files only if they open. Indeed, it is important to model the extent to which the *environment* determines the state after a command. McBride (2011) observes that indexing input and output types over the state effectively lets us specify interfaces in a proof-relevant Hoare logic. Hancock & Hyvernat (2006) have explored the compositionality of indexed "interaction structures", showing that it is possible to model both sharing and independence of state between interfaces.

Session Types as Interface Indices. Our `pipe` is a simple implementation of processes communicating according to a rather unsubtle protocol, with an inevitable but realistic "broken pipe" failure mode. We should surely aim for more sophisticated protocols and tighter compliance. The interface for interaction on a channel should be indexed over session state, ensuring that the requests arriving at a coordinating multihandler match exactly.

Substructural Typing for Honesty with Efficiency. Using `Abort`, we know that the failed computation will not resume under any circumstances, so it is operationally wasteful to construct the continuation. Meanwhile, for `State`, it is usual for the handler to invoke the continuation *exactly once*, meaning that there is no need to allocate space for the continuation in the heap. Moreover, if we want to make promises about the eventual execution of operations, we may need to insist that handlers do invoke continuations sooner or later, and if we want communicating systems to follow a protocol, then they should not be free to drop or resend messages. Linear, affine, and relevant type systems offer tools to manage uses more tightly: we might profitably apply them to continuations and the data structures in which they are stored.

Modules and Type Classes. Frank's effect interfaces provide a form of modularity and abstraction, tailored to effectful programming in direct style. It seems highly desirable to establish the formal status of interfaces with respect to other ways to deliver modularity, such as ML modules (MacQueen, 1984) and Haskell type classes (Wadler & Blott, 1989).

Totality, Productivity, and Continuity. At heart, Frank is a language for incremental transformation of computation (command-response) trees whose node shapes are specified by interfaces, but in the "background", while keeping the values communicated in the foreground. Disciplines for *total* programming over treelike data, as foreground values,

are the staple of modern dependently typed programming languages, with the state of the art continuing to advance (Abel & Pientka, 2013). The separation of client-like inductive structures and server-like coinductive structures is essential to avoid deadlock (e.g. a server hanging) and livelock (e.g. a client constantly interacting but failing to return a value). Moreover, local *continuity* conditions quantifying the relationship between consumption and production (e.g. `spacer` consuming one input to produce two outputs) play a key role in ensuring global termination or productivity. Guarded recursion seems a promising way to capture these more subtle requirements (Atkey & McBride, 2013).

Given that we have the means to negotiate purity locally while still programming in direct style, it would seem a missed opportunity to start from anything other than a not just *pure* but *total* base. To do so, we need to refine our notion of "ability" with a continuity discipline and check that programs obey it, deploying the same techniques total languages use on foreground data for the background computation trees. McBride has shown that general recursion programming fits neatly in a Frank-like setting by treating recursive calls as abstract commands, leaving the semantics of recursion for a handler to determine (McBride, 2015).

11 Conclusion

We have described our progress on the design and implementation of Frank, a language for direct-style programming with locally managed effects. Key to its design is the generalisation of function application to operator application, where an operator is n-adic and may handle effects performed by its arguments. Frank's effect type system statically tracks the collection of permitted effects and convenient syntactic sugar enables lightweight effect polymorphism in which the programmer rarely needs to read or write any effect variables.

It is our hope that Frank can be utilised as a tool for tackling the programming problems we face in real life. Whether we are writing elaborators for advanced programming languages, websites mediating exercises for students, or multi-actor communicating systems, our programming needs increasingly involve the kinds of interaction and control structures which have previously been the preserve of heavyweight operating systems development. It should rather be a joy.

Acknowledgments

We would like to thank the following people: Fred McBride for the idea of generalising functions to richer notions of context; Stevan Andjelkovic, Bob Atkey, James McKinna, Gabriel Scherer, Cameron Swords, and Philip Wadler for helpful feedback; Michael Adams and Adam Gundry for answering questions regarding their respective works and for providing source code used as inspiration; and Daniel Hillerström for guidance on OCaml Multicore. This work was supported by EPSRC grants EP/J014591/1, EP/K034413/1, and EP/M016951/1, a Royal Society Summer Internship, and the Laboratory for Foundations of Computer Science.

Conflict of interest

None.

References

Abel, A. & Pientka, B. (2013) Wellfounded recursion with copatterns: A unified approach to termination and productivity. In ICFP. ACM, pp. 185–196.

Adams, M. D. & Ağacan, Ö. S. (2014) Indentation-sensitive parsing for Parsec. In Haskell. ACM.

Ahman, D. (2017) *Fibred Computational Effects*. Ph.D. thesis, School of Informatics, The University of Edinburgh.

Atkey, R. (2015) *An Algebraic Approach to Typechecking and Elaboration*. https://bentnib.org/posts/2015-04-19-algebraic-approach-typechecking-and-elaboration.html

Atkey, R. & McBride, C. (2013) Productive coprogramming with guarded recursion. In ICFP. ACM, pp. 197–208.

Bauer, A. & Pretnar, M. (2014) An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.* **10**(4), 1–16. https://link.springer.com/chapter/10.1007/978-3-642-40206-7_1

Bauer, A. & Pretnar, M. (2015) Programming with algebraic effects and handlers. *J. log. Alg. Meth. Program.* **84**(1), 108–123.

Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2018) Handle with care: Relational interpretation of algebraic effects and handlers. *PACMPL* **2**(POPL), 8:1–8:30.

Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2019) Abstracting algebraic effects. *PACMPL* **3**(POPL), 1–28.

Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2020) Binders by day, labels by night: Effect instances via lexically scoped handlers. *PACMPL* **4**(POPL), 48:1–48:29.

Brachthäuser, J. I. & Schuster, P. (2017) Effekt: Extensible algebraic effects in Scala (short paper). In SCALA@SPLASH. ACM, pp. 67–72.

Brachthäuser, J. I., Schuster, P. & Ostermann, K. (2018) Effect handlers for the masses. *PACMPL* **2**(OOPSLA), 111:1–111:27.

Brachthäuser, J. I., Schuster, P. & Ostermann, K. (to appear) Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *J. Funct. Program.* **30**.

Brady, E. (2013a) Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* **23**(5), 552–593.

Brady, E. (2013b) Programming and reasoning with algebraic effects and dependent types. In ICFP. ACM, pp. 133–144.

Convent, L. (2017). *Enhancing a Modular Effectful Programming Language*. MSc thesis, School of Informatics, The University of Edinburgh.

Convent, L., Lindley, S., McBride, C. & McLaughlin, C. (2020) Frank repository. Available at: https://www.github.com/frank-lang/frank.

Cooper, E., Lindley, S., Wadler, P. & Yallop, J. (2006) Links: Web programming without tiers. In FMCO. LNCS, vol. 4709. Springer, pp. 266–296.

Dolan, S., White, L., Sivaramakrishnan, K. C., Yallop, J. & Madhavapeddy, A. (2015) Effective concurrency through algebraic effects. In OCaml Workshop.

Dunfield, J. & Krishnaswami, N. R. (2013) Complete and easy bidirectional typechecking for higher-rank polymorphism. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP'13. New York, NY, USA: ACM, pp. 429–442.

Filinski, A. (1999) Representing layered monads. In POPL. ACM, pp. 175–188.

Filinski, A. (2010) Monads in action. In POPL. ACM, pp. 483–494.

Forster, Y., Kammar, O., Lindley, S. & Pretnar, M. (2017) On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *PACMPL* **1**(ICFP), 13:1–13:29.

Forster, Y., Kammar, O., Lindley, S. & Pretnar, M. (2019) On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* **29**, e15.

Fournet, C. & Gonthier, G. (1996) The reflexive CHAM and the join-calculus. In POPL. ACM, pp. 372–385.

Geron, B. (2019) *Defined Algebraic Operations*. Ph.D. thesis, University of Birmingham.

Gundry, A., McBride, C. & McKinna, J. (2010) Type inference in context. In MSFP. ACM.

Gundry, A. M. (2013) *Type Inference, Haskell and Dependent Types*. Ph.D. thesis, University of Strathclyde.

Hancock, P., & Hyvernat, P. (2006) Programming interfaces and basic topology. *Ann. Pure Appl. Logic* **137**(1–3), 189–239.
Hillerström, D. (2015) *Handlers for Algebraic Effects in Links*. MSc thesis, School of Informatics, The University of Edinburgh.
Hillerström, D. (2016) *Compilation of Effect Handlers and their Applications in Concurrency*. MSc(R) thesis, School of Informatics, The University of Edinburgh.
Hillerström, D. & Lindley, S. (2016) Liberating effects with rows and handlers. In TyDe@ICFP. ACM, pp. 15–27.
Hillerström, D., Lindley, S. & Atkey, R. (2020) Effect handlers via generalised continuations. *J. Funct. Program.* **30**, e5.
Huet, G. P. (1997) The zipper. *J. Funct. Program.* **7**(5), 549–554.
Hughes, J. (2004) Programming with arrows. In *Advanced Functional Programming*, Vene, V. & Uustalu, T. (eds). Lecture Notes in Computer Science, vol. 3622. Springer, pp. 73–129.
Inostroza, P. & van der Storm, T. (2018) Jeff: Objects for effect. In Onward! ACM, pp. 111–124.
Jensen, K. & Wirth, N. (1974) *Pascal User Manual and Report*. Berlin, Heidelberg: Springer-Verlag.
Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. In ICFP. ACM, pp. 145–158.
Kiselyov, O. & Ishii, H. (2015) Freer monads, more extensible effects. In Haskell. ACM, pp. 94–105.
Kiselyov, O., Sabry, A. & Swords, C. (2013) Extensible effects: An alternative to monad transformers. In Haskell. ACM, pp. 59–70.
Kmett, E. A. (2015). Trifecta (1.5.2). Available at: http://hackage.haskell.org/package/trifecta-1.5.2.
Leijen, D. (2014) Koka: Programming with row polymorphic effect types. In MSFP. EPTCS, vol. 153, pp. 100–126.
Leijen, D. (2017) Type directed compilation of row-typed algebraic effects. In POPL. ACM, pp. 486–499.
Leijen, D. (2018) *Algebraic Effect Handlers with Resources and Deep Finalization*. Technical report. Microsoft Research.
Leijen, D. & Martini, P. (2015) Parsec (3.1.9). Available at: http://hackage.haskell.org/package/parsec-3.1.9.
Levy, P. B. (2004) *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, vol. 2. Springer.
Lindley, S. (2014) Algebraic effects and effect handlers for idioms and arrows. In WGP@ICFP. ACM, pp. 47–58.
Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. In POPL. ACM, pp. 500–514.
Lucassen, J. M. & Gifford, D. K. (1988) Polymorphic effect systems. In POPL. ACM, pp. 47–57.
MacQueen, D. B. (1984) Modules for standard ML. In LISP and Functional Programming, pp. 198–207.
McBride, C. (2007) How might effectful programs look? In Workshop on Effects and Type Theory. Available at: http://cs.ioc.ee/efftt/mcbride-slides.pdf.
McBride, C. (2011) Kleisli arrows of outrageous fortune. Available at: https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf.
McBride, C. (2012) Frank (0.3). Available at: http://hackage.haskell.org/package/Frank.
McBride, C. (2015) Turing-completeness totally free. In MPC. Lecture Notes in Computer Science, vol. 9129. Springer, pp. 257–275.
McBride, C. (2016). Shonky. Available at: https://github.com/pigworker/shonky.
McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.
Papert, S. (1980) *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Books.
Pierce, B. C. & Turner, D. N. (2000) Local type inference. *ACM Trans. Program. Lang. Syst.* **22**(1), 1–44.
Piróg, M., Polesiuk, P. & Sieczkowski, F. (2019) Typed equivalence of effect handlers and delimited control. In FSCD. LIPIcs, vol. 131. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 30:1–30:16.

Plotkin, G. D. & Power, J. (2001a) Adequacy for algebraic effects. In FoSSaCS. LNCS, vol. 2030. Springer, pp. 1–24.
Plotkin, G. D. & Power, J. (2001b) Semantics for algebraic operations. *Electr. Notes Theor. Comput. Sci.* **45**, 332–345.
Plotkin, G. D. & Power, J. (2002) Notions of computation determine monads. In FoSSaCS. Lecture Notes in Computer Science, vol. 2303. Springer, pp. 342–356.
Plotkin, G. D. & Power, J. (2003) Algebraic operations and generic effects. *Appl. Categ. Struct.* **11**(1), 69–94.
Plotkin, G. D. & Pretnar, M. (2013) Handling algebraic effects. *Log. Methods Comput. Sci.* **9**(4), 1–36.
Pretnar, M. (2009) *The Logic and Handling of Algebraic Effects*. Ph.D. thesis, School of Informatics, The University of Edinburgh.
Pretnar, M. (2014) Inferring algebraic effects. *Log. Methods Comput. Sci.* **10**(3), 1–43.
Schrijvers, T., Piróg, M., Wu, N. & Jaskelioff, M. (2019) Monad transformers and modular algebraic effects: What binds them together. In Haskell@ICFP. ACM, pp. 98–113.
Schrijvers, T., Wu, N., Desouter, B. & Demoen, B. (2014) Heuristics entwined with handlers combined: From functional specification to logic programming implementation. In PPDP. ACM, pp. 259–270.
Shivers, O. & Turon, A. J. (2011) Modular rollback through control logging: A pair of twin functional pearls. In ICFP. ACM, pp. 58–68.
Swamy, N., Guts, N., Leijen, D. & Hicks, M. (2011) Lightweight monadic programming in ML. In ICFP. ACM, pp. 15–27.
Swierstra, W. (2008) Data types à la carte. *J. Funct. Program.* **18**(4), 423–436.
Talpin, J.-P. & Jouvelot, P. (1994) The type and effect discipline. *Inf. Comput.* **111**(2), 245–296.
van der Ploeg, A. & Kiselyov, O. (2014) Reflection without remorse: Revealing a hidden sequence to speed up monadic reflection. In Haskell. ACM, pp. 133–144.
Vonnegut, K. (1982) *Deadeye Dick*. Delacorte.
Wadler, P. (1992) The essence of functional programming. In POPL. ACM, pp. 1–14.
Wadler, P. & Blott, S. (1989) How to make ad-hoc polymorphism less ad-hoc. In POPL. ACM, pp. 60–76.
Wu, N. & Schrijvers, T. (2015) Fusion for free - efficient algebraic effect handlers. In MPC. Lecture Notes in Computer Science, vol. 9129. Springer, pp. 302–322.
Wu, N., Schrijvers, T. & Hinze, R. (2014) Effect handlers in scope. In Haskell. ACM, pp. 1–12.
Zhang, Y. & Myers, A. C. (2019) Abstraction-safe effect handlers via tunneling. *PACMPL* **3**(POPL), 5:1–5:29.

Appendix

Extended proofs

Theorem 5 (Subject Reduction)**.**

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

Proof. By simultaneous induction on the transitions $m \rightsquigarrow_{\mathrm{u}} m'$ and $n \rightsquigarrow_{\mathrm{c}} n'$, and inversion on the typing assumptions. We show some further representative cases of the proof below.

Case R-ASCRIBE-USE.
By inversion of T-SWITCH and T-ASCRIBE, we have $\Phi; \Gamma \,[\Sigma]\vdash\, u \Rightarrow A$ as required.

Case R-HANDLE.

Then by inversion, we have for all cases i,

$$(\Phi \vdash r_{i,j} : \langle \Delta_j \rangle A_j \dashv [\Sigma]\ \Gamma'_{i,j})_j$$

$$\Phi; \Gamma, (\Gamma'_{i,j})_j\ [\Sigma] \vdash\ n_i : B$$

and for all argument positions j,

$$\Sigma \vdash \Delta_j \dashv \Sigma'_j$$

$$\Phi; \Gamma\ [\Sigma'_j] \vdash\ t_j : A_j$$

By T-Ascribe, the obligation reduces to showing:

$$\Phi; \Gamma\ [\Sigma] \vdash\ \overline{\theta}(n_k) : B$$

for minimal k. From R-Handle, we know $(r_{k,j} : \langle \Delta_j \rangle A_j \leftarrow t_j \dashv [\Sigma]\ \theta_j)_j$ so by iterated application of Lemma 4, we have $\Phi \vdash \overline{\theta} : (\Gamma'_{k,j})_j \vDash \Gamma$. The result follows by Lemma 3.

Case R-Ascribe-Cons.
By inversion of T-Switch and T-Ascribe, we have $\Phi; \Gamma\ [\Sigma] \vdash\ w : A$ as required.

Case R-Let.
By inversion of T-Let, we have $P = \forall\, \overline{Z}.A$ and $\Phi; \Gamma\ [\emptyset] \vdash\ w : A$ and $\Phi; \Gamma, f : P\ [\Sigma] \vdash\ n : B$. We have that $\Phi \vdash [\uparrow(w : P)/f] : f : P \vDash \Gamma$ and the result follows by Lemma 3.

Case R-Adapt.
The result follows from inversion and the fact that values are independent of the ambient ability.

Case R-LetRec.
By inversion, we have

- $P_i = \forall\, \overline{Z}.C_i$,
- $(e_i = \overline{\overline{r_i} \mapsto n_i})_i$,
- $(\Phi; \Gamma, \overline{f : P} \vdash e_i : C_i)_i$, and
- $\Phi; \Gamma, \overline{f : P}\ [\Sigma] \vdash\ n : B$.

From these assumptions, we can show that:

$$\overline{\Phi \vdash [\uparrow(\{\overline{r} \mapsto \textbf{letrec}\ \overline{\overline{f : P = e}}\ \textbf{in}\ n\} : P)/f] : \overline{f : P} \vDash \Gamma}$$

holds and the result follows by Lemma 3.

Case R-Freeze-Comm.
By inversion of T-Command and the rule T-Freeze-Cons, noting that

$$\mathsf{af}(c, [\ ]) = s \to s \neq S \to S' \bot$$

for any S, S'.

Case R-Freeze-Frame-Cons.
We have ($\star$) $\Phi; \Gamma\ [\Sigma] \vdash\ \mathscr{F}[\lceil \mathscr{E}[c\ \overline{R}\ \overline{w}] \rceil] : A$ and $\neg(\mathscr{F}[\mathscr{E}]\ \mathsf{handles}\ c)$. We require that

$$\Phi; \Gamma\ [\Sigma] \vdash\ \lceil \mathscr{F}[\mathscr{E}[c\ \overline{R}\ \overline{w}]] \rceil : A$$

for which, by T-Freeze-Cons, it suffices to show

$$\text{(a)}\ \neg(\mathscr{F}[\mathscr{E}]\ \mathsf{handles}\ c) \quad \text{and} \quad \text{(b)}\ \Phi; \Gamma\ [\Sigma] \vdash\ \mathscr{F}[\mathscr{E}[c\ \overline{R}\ \overline{w}]] : A$$

(a) follows by our assumption. For (b), consider the typing of ($\star$) and observe that for any frame $\mathscr{F}$, we have either:

$$\Phi; \Gamma' \; [\Sigma'] \vdash \ulcorner \mathscr{E}[c\ \overline{R}\ \overline{w}] \urcorner \Rightarrow A' \quad \text{or} \quad \Phi; \Gamma' \; [\Sigma'] \vdash \ulcorner \mathscr{E}[c\ \overline{R}\ \overline{w}] \urcorner : A'$$

for some Γ', Σ', and A'. Either case implies the corresponding unfrozen command $\mathscr{E}[c\ \overline{R}\ \overline{w}]$ is well-typed in the frame $\mathscr{F}$ and hence (b) holds. □

Lemma 8. *If* $\Sigma \vdash S : I \dashv \Sigma'; \Omega$ *and* $\Omega \vdash S' : I \dashv \Xi$ *and* $c\ \overline{R} \in (\Xi(\Sigma')@I)(i)$*, then,*

$$\exists j.\ \mathsf{af}(c, I(S \to S'))(i) = j \wedge c\ \overline{R} \in (\Sigma@I)(j).$$

Proof. By induction on the derivation of $\Sigma \vdash S : I \dashv \Sigma'; \Omega$ and case analysis on the derivation of $\Omega \vdash S' : I \dashv \Xi$. The interesting case is I-PAT-BIND:

$$\text{I-PAT-BIND}\ \frac{\Sigma \vdash S : I \dashv \Sigma'; \Omega}{\Sigma, I\ \overline{R} \vdash S\ a : I \dashv \Sigma'; \Omega, a : I\ \overline{R}}$$

Given,

$$\Omega, x : I\ \overline{R'} \vdash S' : I \dashv \Xi \qquad\qquad c\ \overline{R} \in (\Xi(\Sigma')@I)(i)$$

We require to show:

$$\exists j.\ \mathsf{af}(c, I(S\ a \to S'))(i) = j \wedge c\ \overline{R} \in (\Sigma@I, I\ \overline{R'})(j)$$

We perform case analysis on the derivation of $\Omega, x : I\ \overline{R'} \vdash S' : I \dashv \Xi$.

Case I-INST-ID. Therefore, $c\ \overline{R} \in (\Sigma'@I)(i)$. From our inductive hypothesis, there exists j such that:

$$(S \to s)(i) = j \wedge c\ \overline{R} \in (\Sigma@I)(j)$$

The result follows by instantiating the goal with $j + 1$.

Case I-INST-LKP.
We have $c\ \overline{R} \in (\Xi(\Sigma')@I, I\ \overline{R''})(i)$ ($\star$). We perform a case analysis on i.

Case $i = 0$. Then $\overline{R''} = \overline{R}$ and our goal becomes:

$$\exists j.\ \mathsf{af}(c, I(S\ a \to S'\ b))(0) = j \wedge c\ \overline{R} \in (\Sigma@I, I\ \overline{R'})(j)$$

If $a = b$, then instantiate j to 0. Otherwise, $b \in S$ and $b \neq a$. In this case, the result follows by inductive hypothesis with $i = 0$.

Case $i = i' + 1$.
From ($\star$), we have that $c\ \overline{R} \in (\Xi(\Sigma')@I)(i')$. Using this fact to instantiate the inductive hypothesis gives:

$$\exists j.\ (S \to S')(i') = j \wedge c\ \overline{R} \in (\Sigma@I)(j)$$

So we pick $j + 1$ in the goal so it suffices to show:

$$(S\ a \to S'\ b)(i' + 1) = j + 1$$

which holds if $(S'\ b)_{i'+1} = (S\ b)_{j+1}$, i.e. $S'_{i'} = S_j$, which we established from our inductive hypothesis. □

Lemma 9. *If* $\Sigma \vdash \Theta \dashv \Sigma'$ *and* $c\ \overline{R} \in (\Sigma'@I)(i)$*, then,*

$$\exists j.\ \mathsf{af}(c, \Theta)(i) = j \wedge c\ \overline{R} \in (\Sigma@I)(j).$$

Proof. By induction on the derivation of $\Sigma \vdash \Theta \dashv \Sigma'$ using Lemma 8 and the fact that adaptors contain only one component for each interface. □

Lemma 10. *If* $\Sigma \vdash \Xi \dashv \Sigma'$ *and* $c\ \overline{R} \in (\Sigma'@I)(i)$*, then,*

$$\mathsf{af}(c, \Xi)(i) = \bot \vee (\exists j.\ \mathsf{af}(c, \Xi)(i) = j \wedge c\ \overline{R} \in (\Sigma@I)(j)).$$

Proof. By induction on the derivation of $\Sigma \vdash \Xi \dashv \Sigma'$. A-Ext-Id is immediate. For A-Ext-Snoc we have $\Sigma \vdash \Xi, I'\ \overline{R'} \dashv \Sigma', I'\ \overline{R'}$. There are two cases to consider.

Case $I \neq I'$.
Then $(\Sigma', I\ \overline{R'})@I = \Sigma'@I$ and hence $c\ \overline{R} \in (\Sigma'@I)_i$. Also, $\mathsf{af}(c, \Xi, I'\ \overline{R'}) = \mathsf{af}(c, \Xi)$ by definition. The result follows by the inductive hypothesis.

Case $I = I'$.
Then $\mathsf{af}(c, \Xi, I\ \overline{R'}) = (s \to s\ \bot)$, and $(\Sigma', I\ \overline{R'})@I = (\Sigma'@I), I\ \overline{R'}$; hence $c\ \overline{R} \in (\Sigma'@I, I\ \overline{R'})_i$. We perform a case analysis on i. If $i = 0$, it follows $\mathsf{af}(c, \Xi, I\ \overline{R'})(0) = \bot$. Otherwise, we appeal to the inductive hypothesis, picking the appropriate side of the disjunct in each sub-case. □

Corollary 11. *If* $\Sigma \vdash \Delta \dashv \Sigma'$ *and* $c\ \overline{R} \in (\Sigma'@I)(i)$*, then,*

$$\mathsf{af}(c, \Delta)(i) = \bot \vee (\exists j.\ \mathsf{af}(c, \Delta)(i) = j \wedge c\ \overline{R} \in (\Sigma@I)(j)).$$

Proof. By Lemmas 9 and 10 and composition of adaptor functions. □

Theorem 7 (Type Soundness)**.**

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

In particular, if $\Sigma = \emptyset$*, then either the term is a value* w *or the term can reduce by one step.*

Proof. By simultaneous induction on the typing derivations $\cdot; \cdot\ [\Sigma]\vdash m \Rightarrow A$ and $\cdot; \cdot\ [\Sigma]\vdash n : A$. We show some representative cases.

Case T-Ascribe.
By the inductive hypothesis, there are two cases to consider.

Case n is a normal form which respects Σ.

If n is a value, then it is either $\{e\}$, $k\ \overline{w}$ or $\downarrow u$. For the first two cases, it follows that $\uparrow(n : A)$ is a value, hence a normal form. For $n = \downarrow u$, we have $\uparrow(\downarrow u : A) \longrightarrow_{\mathrm{u}} u$ by R-LIFT-UU and R-ASCRIBE-USE using Theorem 5.

Otherwise n is a frozen command $\lceil \mathscr{E}[c\ \overline{R}\ \overline{w}] \rceil$ such that $\mathsf{af}(c, \mathscr{E})(0) = i$ and $c\ \overline{R} \in (\Sigma @ I)(i)$. Since $i \neq \bot$, $\neg(\mathscr{E}\ \mathsf{handles}\ c)$. Then, the term $\uparrow(\lceil \mathscr{E}[c\ \overline{R}\ \overline{w}] \rceil : A)$ takes the form of a frame $\uparrow([\ ] : A)$ plugged with a frozen command. This term reduces by R-FREEZE-FRAME-USE and R-LIFT-UU using the empty evaluation context.

Case there exists a unique $\cdot ; \cdot \ [\Sigma]\vdash\ n' : A$ such that $n \longrightarrow_{\mathrm{c}} n'$.

We have two cases to consider for $n \longrightarrow_{\mathrm{c}} n'$.

Suppose R-LIFT-CC gave rise to the reduction. Then there exist $\mathscr{E}, n_r, n'_r$ such that $n = \mathscr{E}[n_r]$ and $n' = \mathscr{E}[n'_r]$ and $n_r \rightsquigarrow_{\mathrm{c}} n'_r$. Then since n' is well-typed and unique, we can form the unique well-typed term $(\uparrow([\ ] : A))[\mathscr{E}[n'_r]]$ such that $m \longrightarrow_{\mathrm{u}} (\uparrow(\mathscr{E} : A))[n'_r]$ by R-LIFT-CU. The case for R-LIFT-UC is similar.

Case T-APP.

We have three top-level cases to consider: m is a value, m is a frozen command, or m reduces. The latter two cases follow similar reasoning we have shown for the other typing rules, namely appealing to the inductive hypotheses for sub-terms and using the reduction relations on evaluation contexts. We show the case for when m is a value. In particular, $m\ \overline{n}$ takes the form:

$$\uparrow(\{((r_{i,j})_j \to n'_i)_i\} : \{\overline{\langle \Theta \mid \Xi \rangle A \to [\Sigma]B}\})\ \overline{n}$$

for patterns $r_{i,j}$ and terms n'_i. We consider the following cases generated by analysing the form of the argument terms, $\overline{n} = n_1, \ldots, n_l$, given by our inductive hypotheses. From T-APP, we know $\Sigma \vdash \Theta_l \mid \Xi_l \dashv \Sigma'_l$.

Case Suppose the sequence $n_1, \ldots, n_{l-1}$ are all values and n_l is a normal form which respects Σ'_l and let $\mathscr{F} = m\ (n_1, \ldots, n_{l-1}, [\ ])$.

First, if n_l is also a value, then by coverage and our well-typedness assumption on $m\ \overline{n}$ we have at least one clause which matches against the value sequence $\overline{n}$. Let k be the index of the first such clause in m. Then, k is obviously minimal and produces a substitution $\Phi \vdash \theta_k : \Gamma'_k \vDash \Gamma$. Therefore, the result follows by subject reduction (Theorem 5) using R-HANDLE.

Now suppose n_l is a frozen command $\lceil \mathscr{E}[c\ \overline{R}\ \overline{w}] \rceil$ such that $\mathsf{af}(c, \mathscr{E})(0) = i$ and $c\ \overline{R} \in (\Sigma'_l @ I)(i)$. By Corollary 11, there are two cases to consider.

Case $\mathsf{af}(c, \Theta_l \mid \Xi_l)(i) = \bot$.

By definition of $\mathsf{af}(c, \Xi_l)$, we have that $c : \forall \overline{Z}.\overline{A_1} \to B \in \Xi_l$. We perform a case analysis on i. If $i = 0$, then $\mathscr{E}$ $\mathsf{poisedfor}$ c since $\mathsf{af}(c, \mathscr{E}) = (S\ a \to S'\ a)$. Therefore,

$$\mathsf{af}(c, \mathscr{F}); \mathsf{af}(c, \mathscr{E}) = (s \to s\ \bot); (S\ a \to S'\ a)$$

and the result follows by R-HANDLE and R-LIFT-UU.

Suppose, $i = i' + 1$. Therefore, since $\mathsf{af}(c, \mathscr{F}); \mathsf{af}(c, \mathscr{E}) \neq (S \to S'\ \bot)$ we have,

$$\neg((m\ (n_1, \ldots, n_{l-1}, [\ ]))[\mathscr{E}]\ \mathsf{handles}\ c)$$

and the result follows by F-FREEZE-FRAME-USE and R-LIFT-UU.

Case There exists a j s.t. $\mathsf{af}(c, \Theta_l \mid \Xi_l)(i) = j$ and $c\ \overline{R} \in (\Sigma @ I)(j)$.
Then $\mathsf{af}(c, \mathscr{F})$; $\mathsf{af}(c, \mathscr{E}) \neq (S \to S\ \bot)$ and the result follows by F-FREEZE-FRAME-USE and R-LIFT-UU.

Case Suppose the sequence $n_1, \ldots, n_{l-1}$ are all normal forms such that each $n_i = \lceil \mathscr{E}_i[c_i\ \overline{R_i}] \rceil$ respects $\Xi_i(\emptyset)$ and ($\mathscr{E}_i$ poisedfor c_i), and n_l is a normal form.
Observe that the respects condition on the sequence $n_1, \ldots, n_{l-1}$ simply ensures that the sequence will be handled by m since they are either values or they invoke a command in the respective argument extensions. The analysis of n_l follows the same reasoning as the case above and we omit the details.

Case Suppose the sequence $n_1, \ldots, n_{l-1}$ are all normal forms such that each $n_i = \lceil \mathscr{E}_i[c_i\ \overline{R_i}] \rceil$ respects $\Xi_i(\emptyset)$ and ($\mathscr{E}_i$ poisedfor c_i), and n_l reduces.
This case is analogous to the other reduction cases using the appropriate R-LIFT rule.

We have shown three distinct cases when considering the possible forms of the last argument to an application. The same analysis can be performed iteratively starting with the first argument and proceeding left-to-right in evaluation order. Thus, the above analysis suffices to prove this case.

Case T-SWITCH.
By the inductive hypothesis, there are two cases to consider.

Case m is a normal form which respects Σ.
If m is a value, then it must be $\uparrow(v : A)$ for some non-use value v. Hence, we have $\downarrow\uparrow(v : A) \longrightarrow_{\mathrm{c}} v$ by R-ASCRIBE-CONS and R-LIFT-CC using Theorem 5. The case for a frozen command is analogous to T-ASCRIBE and we omit it.

Case There exists a unique $\Phi; \cdot\ [\Sigma]\vdash\ m' \Rightarrow A$ such that $m \longrightarrow_{\mathrm{u}} m'$.
This case is analogous to the corresponding case for T-ASCRIBE.

Case T-COMMAND.
We simplify the analysis by considering the three distinct cases that can occur: all the arguments are values; or there is a left-most frozen argument following some number of value arguments; or there is a left-most redex following some number of value arguments.

Case Suppose our inductive hypotheses assert each $n_i \in \overline{n}$ are all values, $\overline{n} = \overline{w}$.
Then $c\ \overline{R}\ \overline{w} \longrightarrow_{\mathrm{c}} \lceil c\ \overline{R}\ \overline{w} \rceil$ by R-FREEZE-COMM and R-LIFT-CC using Theorem 5.

Case Suppose our inductive hypotheses assert $n_1, \ldots, n_{i-1}$ are all values and n_i is a frozen command $\lceil \mathscr{E}[c'\ \overline{R'}\ \overline{n'}] \rceil$.
This case follows similar reasoning to the corresponding situation in the T-APP case except we need only consider forwarding.

Case Suppose our inductive hypotheses assert $n_1, \ldots, n_{i-1}$ in $\overline{n} = n_1, \ldots, n_k$ are all values, $\overline{w} = w_1, \ldots, w_{i-1}$, and $n_i \longrightarrow_{\mathrm{c}} n_i'$.
We use either R-LIFT-CC or R-LIFT-UC with the frame extension

$$c\ \overline{R}\ (\overline{w}, [\ ], n_{i+1}, \ldots, n_k)$$

depending on the rule used to form $n_i \longrightarrow_{\mathrm{c}} n_i'$.

Case T-DATA.
Similar to the analysis for T-COMMAND.

Case T-THUNK. $\{e\}$ is a non-use value, hence a normal form.

Case T-LET for $\mathbf{let}\, x : P = n\, \mathbf{in}\, n''$.
By the inductive hypothesis for $\cdot; \cdot\, [\emptyset] \vdash n : A$, we have two sub-cases.

Case n is a construction value w.
The result follows by Theorem 5 for R-LET and applying R-LIFT-CC with the empty evaluation context.

Case There exists a unique $\Phi; \cdot\, [\emptyset] \vdash n' : A$ such that $n \longrightarrow_{\mathsf{c}} n'$.
The result follows by either R-LIFT-UC or R-LIFT-CC using the frame extension $\mathbf{let}\, x : P = [\,]\, \mathbf{in}\, n''$.

Case T-LETREC.
By Theorem 5, R-LETREC, and R-LIFT-CC using the empty evaluation context.

Case T-ADAPT.
We have the term $\langle \Theta \rangle n$. The interesting case is when n takes the form of a frozen command, $n = \lceil \mathscr{E}[c\, \overline{R}\, \overline{w}] \rceil$, such that:

$$\mathsf{af}(c, \mathscr{E})(0) = i \qquad\qquad c\, \overline{R} \in (\Sigma @ I)(i)$$

We have that $\neg(\mathscr{E}\ \mathsf{handles}\ c)$. This term reduces by R-FREEZE-FRAME-CONS in the empty evaluation context provided we show: $\neg((\langle \Theta \rangle\ [\,])[\mathscr{E}]\ \mathsf{handles}\ c)$. Which follows since $\mathsf{af}(c, \langle \Theta \rangle); \mathsf{af}(c, \mathscr{E}) \neq S \to S' \bot$ by the definition of the adaptor function at $\langle \Theta \rangle$ and our assumption regarding $\mathsf{af}(c, \mathscr{E})$. □