Do Be Do Be Do

Sam Lindley
The University of Edinburgh, UK
sam.lindley@ed.ac.uk

Conor McBride
University of Strathclyde, UK
conor.mcbride@strath.ac.uk

Craig McLaughlin
The University of Edinburgh, UK
craig.mclaughlin@ed.ac.uk

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 simply 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 avoid mentioning effect variables in source code. This is achieved by propagating an *ambient ability* inwards, rather than accumulating unions of potential effects outwards.

We introduce Frank by example, and then give a formal account of the Frank type system and its semantics. We introduce Core Frank by elaborating Frank operators into functions, case expressions, and unary handlers, and then give a sound small-step operational semantics for Core Frank.

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 system.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords algebraic effects, effect handlers, effect polymorphism, call-by-push-value, pattern matching, continuations, bidirectional typing

1. Introduction

Shall I be pure or impure?

—Philip Wadler [60]

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 Power's algebraic effects [45–48] in conjunction with Plotkin and Pretnar's handlers for algebraic effects [49]—a rich foundation for effectful programming. By separating effect interfaces from their implementation, algebraic effects 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 [34, 57]. 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 simply 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 simply 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 algebraic effects permitted by the current typing context;
- a description of pattern matching compilation from Frank into a fairly standard call-by-value language with unary effect handlers, *Core Frank*;
- a straightforward small-step operational semantics for Core Frank and a proof of type soundness;
- 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 and algebraic effects. Bauer and Pretnar's Eff [6] 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 [5]. Their implementation [50] supports Hindley-Milner type inference and the type system incorporates effect subtyping.

Hillerström and Lindley [19–21] (Links [10]) and Leijen [29] (Koka [28]) have extended existing languages with effect handlers and algebraic effects. 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. [11] built Multicore OCaml by extending OCaml with support for effect handlers and algebraic effects. 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 allow effect variables to be omitted to the degree that Frank does.

Kammar et al. [24] describe a number of effect handler libraries for languages ranging from Racket, to SML, to OCaml, 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 [25, 26] have built another Haskell library for effect handlers, making different design choices.

Brady's `effects` library [7] provides a DSL for programming with effects in the dependently typed language Idris [8]. Like the Haskell libraries, Brady's library currently requires the programmer to write effectful code in a monadic style.

McBride's Shonky [41] is essentially an untyped version of Frank, with a somewhat different concrete syntax. We have built a prototype implementation of Frank by translating typed Frank programs into Shonky. The implementation is available at the following URL:

`https://www.github.com/cmcl/frankjnr`

The rest of the paper is structured as follows. Section 2 introduces Frank by example. Section 3 presents abstract syntax and a type system for Frank. Section 4 describes how to elaborate operators into Core Frank, a language of plain call-by-value functions, explicit case analysis, and unary handler constructs. Section 5 gives an operational semantics for Core Frank and proves type soundness. Section 6 discusses how to store computations in data structures. Section 7 describes the status of our implementation. Section 8 outlines related work. Section 9 discusses future work and Section 10 concludes.

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 [59]

Frank is a functional programming language with effects and handlers in the spirit of Eff [6], controlled by a type system inspired by Levy's call-by-push-value [31]. 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
```

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 (cons x xs) ys = cons 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 (cons x xs) = cons (f x) (map f xs)
```

and apply them in the usual way:

```
map {n -> n+1} (cons 1 (cons 2 (cons 3 nil)))
  = cons 2 (cons 3 (cons 4 nil))
```

A value type `A` is a data type `D R1 ... Rn`, a suspended computation type `{C}`, or a type variable `X`. A computation type resembles a function type `T1 -> ... -> Tm -> [I1 ... In]B` with m *ports* and a *peg* showing the *ability* the computation needs—a bracketed list of n enabled *interfaces*—and the *value type* it delivers. In Frank, names always bind values (a simplifying decision which we shall re-examine in section 9). Top level definitions give names to suspended computations, but we omit the outer braces in their types for convenience.

Type checking separates cleanly into checking the compatibility of value types and checking that required abilities are available. Empty brackets may be omitted. We could have written

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

which really means

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

but have a care: the empty bracket stands for the *ambient* ability, not for purity; the `map` operator is implicitly effect-polymorphic.

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 simulate case-expressions inline using reverse application

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

as in this example of the short-circuited 'and':

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

2.4 Abilities Collect Interfaces; Interfaces Offer Commands

Abilities (Frank's realisation of algebraic effects) are collections of parameterised interfaces, each of which describes a choice of commands (known elsewhere as *operations* [49]). Command types may refer to the parameters of their interface but are not otherwise polymorphic. Here are some simple interfaces.

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

interface Receive X  = receive : X

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

interface Abort      = aborting : Zero
```

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 `set` commands. Note that, unlike data constructors, commands are first-class values. In particular, while `Zero` is uninhabited, `{[Abort]Zero}` contains the value `aborting`. Correspondingly, we can define a polymorphic `abort` which we can use whenever `Abort` is enabled

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

by empty case analysis. The postfix `!` attached to `abort` denotes the definition of a nullary operator.

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 those which permit `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 peg 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—the ambient ability—and where and how that changes.

In designing Frank we have sought to maintain the benefits of effect polymorphism whilst 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 draft of this paper we ruled out explicit effect variables by fiat. But doing so placed artificial restrictions on the formalism (see Section 3), 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 Direct Style for Monadic Programming

We work in a direct applicative style. Where 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! = on 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.

2.6 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.

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 application notation is used for 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" = cons (pair 0 'a')
               (cons (pair 1 'b')
                 (cons (pair 2 'c') nil))
```

What is happening?

The type of `state` shows us that Frank operators do not merely have input *types*, but input *ports*, specifying not only the types of the values expected, 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 port using only that ability; the ambient ability at its second port 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* arrives on the second port. 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 arrive at that port. In particular, *requests* can arrive at the second port, 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 ports.

Having received signals for each argument the `state` operator should *handle* them. If the input on the second port is a value, then that value is returned. If the input on the second port 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 port and the continuation *invoked* in the second port.

We emphasise a key difference between Frank and most other languages supporting algebraic effect handlers (including Eff, Koka, and Multicore OCaml [11]): 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 s 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 algebraic 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. [24] provide a more in-depth discussion of the trade-offs between deep and shallow handlers.

2.7 Handling on Multiple Ports

Frank allows the programmer to write n-ary operators, so we can offer different adjustments to the ambient ability at different ports. 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 port and all commands from `Receive X` on its second port. We say that `pipe` is thus a *multihandler*. The first argument has type `Unit` and

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 (cons "do" (cons "be" (cons "" nil))))
     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 (cons "do" (cons "be" (cons "" nil))))
     (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 (cons "do" (cons "be" (cons "" nil))))
      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. [24] 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 non-trivial 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 on a port promises *exactly* what will be handled at that port. The peg of `pipe` requires the ambient ability to support `Abort`, and its ports 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 port 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 at the relevant port, with other commands forwarded transparently to the most local port 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 port 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 ports and pegs. 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 type of the port has been polluted by the ability of the peg and it now fails to advertise precisely which interfaces it handles.

2.8 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 <aborting -> _> 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 [28].

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 `aborting` only in its first port, 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 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. Just like Koka, Frank resolves any replication of interfaces by shadowing, discarding all but the last instance of an interface in an ability.

2.9 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 [5, 24].

Frank computations are abstract: a thing of type `{C}` can be communicated or invoked, but not inspected. Ports 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 Sections 4 and 5.

However, let us first give the type system for these programs and show how Frank's careful silences deliver the power we claim.

3. A Frank Formalism

A value is. A computation does.

—Paul Blain Levy [31]

In this section we give a formal presentation of the abstract syntax and type system of Frank.

Types

(value types)	$A, B ::= D\ \overline{R} \mid \{C\} \mid X$
(computation types)	$C ::= \overline{T \to} G$
(ports)	$T ::= \langle\Delta\rangle A$
(pegs)	$G ::= [\Sigma]A$
(type variables)	$Z ::= X \mid [E]$
(type arguments)	$R ::= A \mid [\Sigma]$
(polytypes)	$P ::= \forall \overline{Z}.A$
(abilities)	$\Sigma ::= \varnothing \mid \Sigma, I\ \overline{R} \mid E$
(adjustments)	$\Delta ::= \iota \mid \Delta + I\ \overline{R}$
(type environments)	$\Gamma ::= \cdot \mid \Gamma, x : A \mid f : P$

Terms

(uses)	$m ::= x \mid f \mid c \mid m\ s$
(constructions)	$n ::= m \mid k\ \overline{n} \mid \{e\}$
	$\mid\ \mathbf{let}\ f : P = n\ \mathbf{in}\ n'$
	$\mid\ \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n$
(spines)	$s ::= \overline{n}$
(computations)	$e ::= \overline{\overline{r} \mapsto n}$
(computation patterns)	$r ::= p \mid \langle c\ \overline{p} \to z\rangle \mid \langle x\rangle$
(value patterns)	$p ::= k\ \overline{p} \mid x$

Figure 1. Frank Abstract Syntax

3.1 Syntax

The abstract syntax of Frank is given in Figure 1.

The types are divided into value types and computation types. Value types are data types ($D\ \overline{R}$), suspended computation types ($\{C\}$), or type variables (X).

Computations types are build from input *ports* T and output *pegs* G. A computation type

$$C = \langle\Delta_1\rangle A_1 \to \cdots \to \langle\Delta_n\rangle A_n \to [\Sigma]B$$

has ports $\langle\Delta_1\rangle A_1, \ldots, \langle\Delta_n\rangle A_n$ and peg $[\Sigma]B$. A computation of type C must handle effects in Δ_i on the i-th argument. All arguments are handled simultaneously. As a result it returns a value of type B and may perform effects in Σ.

A port $\langle\Delta\rangle A$ constrains an input. The adjustment Δ describes the difference between the ambient effects and the effects of the input, in other words, those effects occurring in the input that must be handled on that port. A peg $[\Sigma]A$ constrains an output. The effects Σ are those that result from running the computation.

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

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

Modulo the braces around the function type, 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:

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

We adopt the convention that the identity adjustment ι may be omitted from adjustments and ports.

$$I_1\ \overline{R_1}, \ldots, I_n\ \overline{R_n} \equiv \iota + I_1\ \overline{R_1} + \cdots + I_n\ \overline{R_n}$$
$$A \equiv \langle\iota\rangle A$$

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

$$I_1\ \overline{R_1}, \ldots, I_n\ \overline{R_n} \equiv \varepsilon, I_1\ \overline{R_1}, \ldots, I_n\ \overline{R_n}$$
$$A \equiv [\varepsilon]A$$

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 anywhere in a Frank program.

We let X range over ordinary type variables and E range over effect variables; polytypes may be polymorphic in both. Though we avoid effect variables in source code, we are entirely explicit about them in the abstract syntax and the type system.

Data Types and effect interfaces are defined globally. A definition for data type $D(\overline{Z})$ consists of a collection of data constructor signatures of the form $k : \overline{A}$, where the type/effect variables $\overline{Z}$ may be bound in the data constructor arguments $\overline{A}$. Each data constructor belongs to a single data type and may appear only once in that data type. We write $\mathcal{D}(D\ \overline{R}, k)$ for the type arguments of constructor k of data type $D\ \overline{R}$. A definition for effect interface $I(\overline{Z})$ consists of a collection of command signatures of the form $c : \overline{A} \to B$, denoting that command c takes arguments of types $\overline{A}$ and returns a value of type B. The types $\overline{A}$ and B may all depend on $\overline{Z}$. Each command belongs to a single interface and may appear only once in that interface. We write $\mathcal{I}(I\ \overline{R}, c)$ for the signature of command c of effect interface $I\ \overline{R}$

Effect Parameters with an Invisible Effect Variable In the case that the first parameter of a data type or effect interface definition is its only effect variable ε, then we may omit it from the definition (we give an example in Section 6).

An ability is a collection of interfaces initiated either with the empty ability $\varnothing$ (yielding a *closed* ability) or an effect variable E (yielding an *open* ability). Order is important, as repeats are permitted, in which case the right-most interface overrides all others with the same name. Closed abilities are not normally required, but they can be used to enforce purity, for instance. In ASCII source code we write $\varnothing$ as `0`.

Adjustments modify abilities. The identity adjustment ι leaves an ability unchanged. An adjustment $\Delta + I\ \overline{R}$ extends an ability with the interface $I\ \overline{R}$. The action of an adjustment Δ on an ability Σ is given by the $\oplus$ operation.

$$\Sigma \oplus \iota = \Sigma$$
$$\Sigma \oplus (\Delta + I\ \overline{R}) = (\Sigma \oplus \Delta), I\ \overline{R}$$

Type environments distinguish monomorphic and polymorphic variables.

Frank follows a bidirectional typing discipline [44]. Thus terms are subdivided into *uses* whose type may be inferred, and *constructions* which may be checked against a type. Uses comprise monomorphic variables (x), polymorphic variables (f), commands (c), and applications ($m\ s$). Constructions comprise uses (m), data constructor instances ($k\ \overline{n}$), suspended computations ($\{e\}$), polymorphic let ($\mathbf{let}\ f : P = n\ \mathbf{in}\ n'$) and mutual recursion ($\mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n$). A spine ($s$) is a sequence of constructions ($\overline{n}$). We write ! for the empty spine.

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 binding its arguments to $\overline{p}$ and 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.

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

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

$$\textsc{Var}\quad \frac{x : A \in \Gamma}{\Gamma\,[\Sigma]\vdash x \Rightarrow A} \qquad \textsc{PolyVar}\quad \frac{f : \forall \overline{Z}.A \in \Gamma}{\Gamma\,[\Sigma]\vdash f \Rightarrow \theta(A)} \qquad \textsc{Command}\quad \frac{c : \overline{A \to} B \in \Sigma}{\Gamma\,[\Sigma]\vdash c \Rightarrow \{\overline{\langle\iota\rangle A \to}\,[\Sigma]B\}}$$

$$\textsc{App}\quad \frac{\Gamma\,[\Sigma]\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to}\,[\Sigma']B\} \qquad \Sigma' = \Sigma \qquad \overline{\Gamma\,[\Sigma \oplus \Delta]\vdash n : A}}{\Gamma\,[\Sigma]\vdash m\,\overline{n} \Rightarrow B}$$

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

$$\textsc{Switch}\quad \frac{\Gamma\,[\Sigma]\vdash m \Rightarrow A \qquad A = B}{\Gamma\,[\Sigma]\vdash m : B} \qquad \textsc{Data}\quad \frac{k\,\overline{A} \in D\,\overline{R} \qquad \overline{\Gamma\,[\Sigma]\vdash n : A}}{\Gamma\,[\Sigma]\vdash k\,\overline{n} : D\,\overline{R}} \qquad \textsc{Thunk}\quad \frac{\Gamma \vdash e : C}{\Gamma\,[\Sigma]\vdash \{e\} : \{C\}}$$

$$\textsc{Let}\quad \frac{P = \forall \overline{Z}.A \qquad \Gamma\,[\Sigma]\vdash n : A \qquad \Gamma, f : P\,[\Sigma]\vdash n' : B}{\Gamma\,[\Sigma]\vdash \mathbf{let}\ f : P = n\ \mathbf{in}\ n' : B} \qquad \textsc{LetRec}\quad \frac{\overline{P} = \forall \overline{Z}.\{C\} \qquad \overline{\Gamma, \overline{f : P} \vdash e : C} \qquad \Gamma, \overline{f : P}\,[\Sigma]\vdash n : B}{\Gamma\,[\Sigma]\vdash \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n : B}$$

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

$$\textsc{Comp}\quad \frac{(r_{i,j} : T_j \dashv [\Sigma]\ \Gamma'_{i,j})_{i,j} \qquad (\Gamma, (\Gamma'_{i,j})_j\,[\Sigma]\vdash n_i : B)_i \qquad (r_{i,j})_{i,j}\ \text{covers}\ (T_j)_j}{\Gamma \vdash ((r_{i,j})_j \mapsto n_i)_i : (T_j \to)_j\,[\Sigma]B}$$

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

$$\textsc{P-Var}\quad \frac{}{x : A \dashv x : A} \qquad \textsc{P-Data}\quad \frac{k\,\overline{A} \in D\,\overline{R} \qquad \overline{p : A \dashv \Gamma}}{k\,\overline{p} : D\,\overline{R} \dashv \overline{\Gamma}}$$

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

$$\textsc{P-Value}\quad \frac{p : A \dashv \Gamma}{p : \langle\Delta\rangle A \dashv [\Sigma]\ \Gamma} \qquad \textsc{P-Request}\quad \frac{c : \overline{A} \to B \in \varnothing \oplus \Delta \qquad (p_i : A_i \dashv \Gamma_i)_i}{\langle c\,\overline{p} \to z\rangle : \langle\Delta\rangle B' \dashv [\Sigma]\ \overline{\Gamma}, z : \langle\iota\rangle B \to [\Sigma \oplus \Delta]B'} \qquad \textsc{P-CatchAll}\quad \frac{}{\langle x\rangle : \langle\Delta\rangle A \dashv [\Sigma]\ x : \{[\Sigma \oplus \Delta]A\}}$$

Figure 2. Frank Typing Rules

from Section 2.

```
letrec map :
  ∀ε X Y.{⟨ι⟩{⟨ι⟩X → [ε]Y} → ⟨ι⟩(List X) → [ε](List Y)}
    = f nil            ↦ nil
      f (cons x xs)    ↦ cons (f x) (map f xs) in
letrec state : ∀ε X.{⟨ι⟩X → ⟨ι + State S⟩X → [ε]X}
    =  s x             ↦ x
       s ⟨get ↦ k⟩     ↦ state s (k s)
       s ⟨set s' ↦ k⟩  ↦ state s' (k unit) in
let index : ∀ε X.{⟨ι⟩List X → [ε]List (Pair Nat X)} =
    = {xs ↦ state zero (map {x ↦ pair next! x} xs)} in
index "abc"
```

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.

3.2 Typing Rules

The typing rules for Frank are given in Figure 2. The inference judgement $\Gamma\,[\Sigma]\vdash m \Rightarrow A$ states that in type environment Γ with ambient ability Σ, we can infer that use m has type A. The checking judgement $\Gamma\,[\Sigma]\vdash n : A$ states that in type environment Γ with ambient ability Σ, construction n has type A. The auxiliary judgement $\Gamma \vdash e : C$ states that in type environment Γ, computation e has type C. The judgement $r : T \dashv [\Sigma]\ \Gamma$ states that computation pattern r of port type T with ambient ability Σ binds type environment Γ. The judgement $p : A \dashv \Gamma$ states that value pattern p of type A binds type environment Γ.

The VAR rule infers the type of a monomorphic variable x by looking it up in the environment; POLYVAR does the same for a polymorphic variable f, but also instantiates type variables and effect variables through substitution θ: the presentation is declarative, so θ is unconstrained. The COMMAND rule infers the type of a command c by looking it up in the ambient ability, where the ports have the identity adjustment and the peg has the ambient ability.

The APP rule infers the type of an application $m\ \overline{n}$ under ambient ability Σ. First it infers the type of m of the form $\{\overline{\langle\Delta\rangle A} \to [\Sigma']B\}$. Then it checks that $\Sigma' = \Sigma$ and that each argument n_i matches the inferred type in the ambient ability Σ extended with adjustment Δ_i. If these checks succeed, then the inferred type for the application is B.

The SWITCH rule allows us to treat a use as a construction. The checking rules for data types (DATA), suspended computations (THUNK), polymorphic let (LET), and mutual recursion (LETREC) recursively check the subterms.

Notation We write $(M)_i$ for a list of zero or more copies of M indexed by i. Similarly, we write $(M)_{i,j}$ for a list of zero or more copies of M indexed by i and j.

A computation of type $\overline{T \to} \; G$ is built by composing pattern matching clauses of the form $\overline{r} \mapsto n$ (COMP), where $\overline{r}$ is a sequence of computation patterns whose variables are bound in n. The side condition in the COMP rule requires that the patterns in the clauses cover all possible values inhabiting the types of the ports. Pattern elaboration (Section 4) yields an algorithm for checking coverage.

Value patterns can be typed as computation patterns (P-VALUE). A request pattern $\langle c\,\overline{p} \to z\rangle$ may be checked at type $\langle\Delta\rangle B'$ with ambient ability Σ (P-REQUEST). The command c must be in the adjustment Δ. The continuation is a plain function so its port type has the identity adjustment. The continuation's peg has the ambient ability with Δ applied. To check a computation pattern $\langle x\rangle$ we apply the adjustment to the ambient ability (P-CATCHALL).

Instantiating ε In an earlier draft of this paper, 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 ambients match up exactly where needed, relaxing the POLYVAR rule does no harm, and now we can support data types parameterised by multiple effect variables.

Subeffecting One strength of bidirectional type systems [44] is how smoothly they extend to support subtyping rules. Before adopting operators, we considered incorporating a subeffecting judgement. But, in the presence of operators, subeffecting does not seem particularly helpful, as operators are invariant in their effects.

4. Core Frank

We elaborate Frank into Core Frank, a language in which operators are implemented through a combination of call-by-value functions, case statements, and unary effect handlers. Operators in Frank elaborate to n-ary functions over suspended computations in Core Frank. Shallow pattern matching on a single computation elaborates to unary effect handling. Shallow pattern matching on a data type value elaborates to case analysis. Deep pattern matching on multiple computations elaborates to a tree of unary effect handlers and case statements.

The abstract syntax of Core Frank is given in Figure 3. The only difference between Frank and Core Frank types is that a computation type in Frank takes n ports to a peg, whereas a computation type in Core Frank takes n value types to a peg. The difference between the term syntaxes is more significant. Polymorphic type variables are instantiated explicitly with type arguments. Constructions may be coerced to uses via a type annotation, which is helpful for the semantics (Section 5), where constructions are often substituted for variables. In place of pattern-matching suspended computations, we have n-argument lambda abstractions, case statements, and unary effect handlers. The first two abstractions are standard; the third eliminates a single effectful computation. Elimination of commands is specified by command clauses which arise from request and catch-all patterns in the source language. Elimination of return values is specified by the single return clause, which arises from value patterns in the source language. The adjustment annotation is necessary for type checking. As we no longer have operators, application is now plain n-ary call-by-value function application.

The Core Frank typing rules are given in Figure 4. They are mostly unsurprising given the corresponding Frank Typing rules. The HANDLE rule requires the adjustment Δ for its premisses (the source language builds this adjustment into ports). It is also annotated with the result type. The COERCE rule allows constructions to be treated as uses. The type annotations in the HANDLE and COERCE rules are used in the operational semantics.

Types

(value types)	$A, B ::= D\,\overline{R} \mid \{C\} \mid X$
(computation types)	$C ::= \overline{A \to} \; G$
(pegs)	$G ::= [\Sigma]A$
(type variables)	$Z ::= X \mid [E]$
(type arguments)	$R ::= A \mid [\Sigma]$
(polytypes)	$P ::= \forall\overline{Z}.A$
(abilities)	$\Sigma ::= \varnothing \mid \Sigma, I\,\overline{R} \mid E$
(adjustments)	$\Delta ::= \iota \mid \Delta + I\,\overline{R}$
(type environments)	$\Gamma ::= \cdot \mid \Gamma, x : A \mid f : P$

Terms

(uses)	$m ::= x \mid f\,\overline{R} \mid c \mid m\,s \mid (n : A)$
(constructions)	$n ::= m \mid k\,\overline{n}$
	$\mid \lambda\overline{x}.n$
	$\mid \textbf{case}\; m \;\textbf{of}\; \overline{k\,\overline{x} \mapsto n}$
	$\mid \textbf{handle}^{\Delta}_{G}\; m \;\textbf{with}\; \overline{c\,\overline{x} \to z \mapsto n} \mid x \mapsto n'$
	$\mid \textbf{let}\; f : P = n \;\textbf{in}\; n'$
	$\mid \textbf{letrec}\; \overline{f : P = \lambda\overline{x}.n} \;\textbf{in}\; n'$
(spines)	$s ::= \overline{n}$

Figure 3. Core Frank Abstract Syntax

Notation We extend our indexed list notation to allow indexing over data constructors and commands. In typing rules, we follow the convention that if a meta variable appears only inside such an indexed list then it is implicitly indexed. For instance, the n in the CASE rule depends on k, whereas Σ does not because it appears outside as well as inside an indexed list.

4.1 Elaboration

We now describe the elaboration of Frank into Core Frank by way of a translation $[\![-]\!]$.

We begin with the translation on types. Most cases are homomorphic, that is, given by structurally recursive boilerplate, so we only give the non-trivial cases. In order to translate a computation type we supply the ambient ability to each of the ports.

$$[\![\overline{T \to}\; [\Sigma]A]\!] = \overline{[\![T]\!]([\![\Sigma]\!]) \to}\; [[\![\Sigma]\!]][\![A]\!]$$

Each port elaborates to a suspended computation type with effects given by the ambient ability extended by the adjustment at the port.

$$[\![\langle\Delta\rangle A]\!](\Sigma) = \{[\Sigma \oplus [\![\Delta]\!]][\![A]\!]\}$$

The translation on terms depends on the type of the term so we specify it as a translation on derivation trees. As with the translation on types, we give only the non-trivial cases: all of the other cases are homomorphic. The translation on polymorphic variables converts implicit instantiation into explicit type application.

$$\left[\!\!\left[\frac{f : \forall\overline{Z}.A \in \Gamma}{\Gamma\ [\Sigma]\!\vdash f \Rightarrow \theta(A)}\right]\!\!\right] = \frac{f : \forall\overline{Z}.[\![A]\!] \in [\![\Gamma]\!]}{\Gamma\ [\Sigma]\!\vdash f\,\overline{R} \Rightarrow [\![A]\!][\overline{R}/\overline{Z}]},$$

where $R_i = [\![\theta(Z_i)]\!]$ and $A[\overline{R}/\overline{Z}]$ denotes the simultaneous substitution of each type argument R_i for type variable Z_i in value type A. In the remaining non-trivial cases, we save space by writing only the judgement at the root of a derivation and by writing only the term when referring to a descendent of the root.

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

$$\textsc{Var}\quad \frac{x : A \in \Gamma}{\Gamma\,[\Sigma]\vdash x \Rightarrow A} \qquad \textsc{PolyVar}\quad \frac{f : P \in \Gamma}{\Gamma\,[\Sigma]\vdash f\,\overline{R} \Rightarrow P(\overline{R})} \qquad \textsc{Command}\quad \frac{c : \overline{A \to}\, B \in \Sigma}{\Gamma\,[\Sigma]\vdash c \Rightarrow \{\overline{A \to}\,[\Sigma]B\}}$$

$$\textsc{App}\quad \frac{\Gamma\,[\Sigma]\vdash m \Rightarrow \{\overline{A \to}\,[\Sigma']B\} \qquad \Sigma' = \Sigma \qquad \overline{\Gamma\,[\Sigma]\vdash n : A}}{\Gamma\,[\Sigma]\vdash m\,\overline{n} \Rightarrow B} \qquad \textsc{Coerce}\quad \frac{\Gamma\,[\Sigma]\vdash n : A}{\Gamma\,[\Sigma]\vdash (n : A) \Rightarrow A}$$

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

$$\textsc{Switch}\quad \frac{\Gamma\,[\Sigma]\vdash m \Rightarrow A \qquad A = B}{\Gamma\,[\Sigma]\vdash m : B} \qquad \textsc{Data}\quad \frac{k\,\overline{A} \in D\,\overline{R} \qquad \overline{\Gamma\,[\Sigma]\vdash n : A}}{\Gamma\,[\Sigma]\vdash k\,\overline{n} : D\,\overline{R}} \qquad \textsc{Fun}\quad \frac{\Gamma, \overline{x} : \overline{A}\,[\Sigma']\vdash n : B}{\Gamma\,[\Sigma]\vdash \lambda\overline{x}.n : \{\overline{A \to}\,[\Sigma']B\}}$$

$$\textsc{Case}\quad \frac{\Gamma\,[\Sigma]\vdash m \Rightarrow D\,\overline{R} \qquad (\Gamma, \overline{x} : \overline{A}\,[\Sigma]\vdash n : B)_{k\,\overline{A} \in D\,\overline{R}}}{\Gamma\,[\Sigma]\vdash \mathbf{case}\; m\; \mathbf{of}\; (k\,\overline{x} \mapsto n)_k : B}$$

$$\textsc{Handle}\quad \frac{\Gamma\,[\Sigma \oplus \Delta]\vdash m \Rightarrow A' \qquad (\Gamma, \overline{x} : \overline{A}, z : B \to [\Sigma]A'\,[\Sigma]\vdash n : B')_{c : \overline{A \to}\; B \,\in\, \varnothing \oplus \Delta} \qquad \Gamma, x : A'\,[\Sigma]\vdash n' : B'}{\Gamma\,[\Sigma]\vdash \mathbf{handle}^{\Delta}_{[\Sigma]B'}\; m\; \mathbf{with}\; (c\,\overline{x} \to z \mapsto n)_c \mid x \mapsto n' : B'}$$

$$\textsc{Let}\quad \frac{P = \forall\overline{Z}.A \qquad \Gamma\,[\Sigma]\vdash n : A \qquad \Gamma, f : P\,[\Sigma]\vdash n' : B}{\Gamma\,[\Sigma]\vdash \mathbf{let}\; f : P = n\; \mathbf{in}\; n' : B} \qquad \textsc{LetRec}\quad \frac{\overline{P = \forall\overline{Z}.\{C\}} \qquad \overline{\Gamma, \overline{f : P}\,[\Sigma]\vdash \lambda\overline{x}.n : \{C\}} \qquad \Gamma, \overline{f : P}\,[\Sigma]\vdash n' : B}{\Gamma\,[\Sigma]\vdash \mathbf{letrec}\; \overline{f : P = \lambda\overline{x}.n}\; \mathbf{in}\; n' : B}$$

Figure 4. Core Frank Typing Rules

The crux of the translation is the elaboration of pattern matching for computations. Computations can occur either in suspended computations or mutually recursive definitions. In order to translate a computation we supply the ambient ability.

$$\begin{aligned}
&[\![\Gamma\,[\Sigma]\vdash \{e\} : C]\!] = [\![e]\!]([\![\Sigma]\!])\\
&[\![\Gamma\,[\Sigma]\vdash \mathbf{letrec}\; \overline{f : P = e}\; \mathbf{in}\; n : B]\!] =\\
&\qquad [\![\Gamma]\!]\,[\![\Sigma]\!]\vdash \mathbf{letrec}\; \overline{f : [\![P]\!] = [\![e]\!]([\![\Sigma]\!])}\; \mathbf{in}\; [\![n]\!]
\end{aligned}$$

The translation of a computation is then defined as follows

$$\begin{aligned}
&[\![\Gamma \vdash ((r_{i,j})_j \mapsto n_i)_i : (T_j \to)_j\; G]\!](\Sigma) =\\
&\qquad [\![\Gamma]\!]\,[\Sigma]\vdash \lambda(x_j)_j.PE((x_j)_j, (T_j)_j, ((r_{i,j})_j \mapsto [\![n_i]\!])_i, G)\\
&\qquad\qquad : \{([\![T_j]\!] \to)_j\; [\![G]\!]\}
\end{aligned}$$

where each x_j is fresh and $PE(\overline{x}, \overline{Q}, \overline{u}, G)$ is a function that takes a list of variables $\overline{x}$ to eliminate, a list of pattern types $\overline{Q}$, a pattern matching matrix $\overline{u}$, and a result type G, and yields a Core Frank term. Pattern types (Q) are either value types (A), port types (T), or inscrutable types ($\bullet$). We use the latter to avoid trying to reconstruct continuation types. A pattern matching matrix $\overline{u}$ is a list of pattern matching clauses, where the body n of each clause $u = \overline{r} \mapsto n$ is a Core Frank construction instead of a source Frank construction.

The pattern matching elaboration function PE is defined in Figure 5. For this purpose we find it convenient to use functional programming list notation. We write $[\,]$ for the empty list, $v :: vs$ for the list obtained by prepending element v to the beginning of the list vs, $[v]$ as shorthand for $v :: [\,]$, and $vs +\!\!+ ws$ for the list obtained by appending list ws to the end of list vs. There are four cases for PE. If the head pattern type is a data type, then it generates a case split. If the head pattern is a port type then it generates a handler. If the head pattern is some other pattern type (a suspended computation type or a type variable) then neither eliminator is produced. If the lists are empty then the body of the head clause is returned.

Sequencing Computations We write $\mathbf{let}\; x = n\; \mathbf{in}\; n'$ as syntactic sugar for $on\; n\; \{x \mapsto n'\}$, where on is as in Section 2:

$$\begin{aligned}
&\mathbf{let}\; x = n\; \mathbf{in}\; n' \equiv\\
&\quad \mathbf{let}\; (on : \forall\varepsilon\; X\; Y.\{\langle\iota\rangle X \to \langle\iota\rangle\{\langle\iota\rangle X \to [\varepsilon]Y\} \to [\varepsilon]Y\}) =\\
&\qquad \{x\; f \;\mapsto\; f\; x\}\; \mathbf{in}\; on\; n\; \{x \mapsto n'\}
\end{aligned}$$

This sugar differs from the polymorphic let construct in two ways: 1) it has no type annotation on n, and 2) x is monomorphic in n'.

We make use of several auxiliary functions. The $Patterns$ function returns a complete list of patterns associated with the supplied data type or interface. The $PatternTypes$ function takes a data type and constructor or interface and command, and returns a list of types of the components of the constructor or command. The operation $us\; @\; r$ projects out a new pattern matching matrix from us filtered by matching the pattern r against the first column of us. We make use of the obvious generalisations of let binding for binding multiple constructions and for rebinding patterns.

Example To illustrate how operators are elaborated into Core Frank, we give the Core Frank representation of the `pipe` multi-handler defined in Section 2.7.

$$\begin{aligned}
&\mathbf{letrec}\; \mathsf{pipe} :\\
&\quad \forall\varepsilon\; X\; Y.\{\{[\varepsilon, \mathsf{Abort}, \mathsf{Send}\; X]\mathsf{Unit}\} \to\\
&\qquad\qquad \{[\varepsilon, \mathsf{Abort}, \mathsf{Receive}\; X]Y\} \to [\varepsilon, \mathsf{Abort}]Y\}\\
&\qquad = \lambda x\; y.\mathbf{handle}^{\iota + \mathsf{Send}\; X}_{[\varepsilon, \mathsf{Abort}]Y}\; x!\; \mathbf{with}\\
&\qquad\qquad \langle \mathsf{send}\; \mathsf{x} \to \mathsf{s}\rangle \mapsto\\
&\qquad\qquad\quad \mathbf{handle}^{\iota + \mathsf{Receive}\; X}_{[\varepsilon, \mathsf{Abort}]Y}\; y!\; \mathbf{with}\\
&\qquad\qquad\qquad \langle \mathsf{receive} \to \mathsf{r}\rangle \mapsto \mathsf{pipe}\; (s\; \mathsf{unit})\; (r\; x)\\
&\qquad\qquad\qquad y \qquad\qquad\quad \mapsto y\\
&\qquad\qquad x \mapsto \mathbf{case}\; x\; \mathbf{of}\; \mathsf{unit} \mapsto\\
&\qquad\qquad\qquad\quad \mathbf{handle}^{\iota + \mathsf{Receive}\; X}_{[\varepsilon, \mathsf{Abort}]Y}\; y!\; \mathbf{with}\\
&\qquad\qquad\qquad\qquad \langle \mathsf{receive} \to \mathsf{r}\rangle \mapsto \mathsf{abort}!\\
&\qquad\qquad\qquad\qquad y \qquad\qquad\quad \mapsto y
\end{aligned}$$

$$
\begin{aligned}
PE(x :: xs, D\ Rs :: Qs, us, G) &= \mathbf{case}\ x\ \mathbf{of} \\
&\quad (k_i\ ys_i \mapsto PE(ys_i +\!\!+ xs, PatternTypes(D\ Rs, k_i) +\!\!+ Qs, us\ @\ k_i\ ys_i, G))_i \\
&\quad \text{where} \\
&\quad (k_i\ ys_i)_i = Patterns(D) \\
PE(x :: xs, \langle\Delta\rangle A :: Qs, us, G) &= \mathbf{handle}^{[\![\Delta]\!]}_{[\![G]\!]}\ x!\ \mathbf{with} \\
&\quad (c_i\ ys_i \to z_i \mapsto PE(z_i :: ys_i +\!\!+ xs, PatternTypes(\Delta, c_i) +\!\!+ Qs, us\ @\ c_i\ ys \to z_i, G))_i \\
&\quad w \mapsto PE(w :: xs, A :: Qs, us, G) \\
&\quad \text{where} \\
&\quad w :: (\langle c_i\ ys_i \to z_i\rangle)_i = Patterns(\Delta) \\
PE(x :: xs, Q :: Qs, us, G) &= PE(xs, Qs, us\ @\ x, G) \\
PE([], [], ([] \mapsto n) :: us, G) &= n
\end{aligned}
$$

$$
\begin{aligned}
[]\ @\ r &= [] \\
(u :: us)\ @\ r &= (u\ @\ r) :: (us\ @\ r)
\end{aligned}
$$

$$
\begin{aligned}
(k\ ps' :: rs \mapsto n)\ @\ k\ ps &= [ps +\!\!+ rs \mapsto \mathbf{let}\ ps' = ps\ \mathbf{in}\ n] \\
(k'\ ps' :: rs \mapsto n)\ @\ k\ ps &= [], \quad \text{if } k \neq k' \\
(x :: rs \mapsto n)\ @\ k\ ps &= [ps +\!\!+ rs \mapsto \mathbf{let}\ x = k\ ps\ \mathbf{in}\ n] \\
(\langle x\rangle :: rs \mapsto n)\ @\ k\ ps &= [ps +\!\!+ rs \mapsto \mathbf{let}\ x = \{k\ ps\}\ \mathbf{in}\ n] \\
(r :: rs \mapsto n)\ @\ k\ ps &= []
\end{aligned}
$$

$$
\begin{aligned}
(\langle c\ ps' \to q'\rangle :: rs \mapsto n)\ @\ \langle c\ ps \to q\rangle &= [q :: ps +\!\!+ rs \mapsto \mathbf{let}\ (q' :: ps') = (q :: ps)\ \mathbf{in}\ n] \\
(\langle x\rangle :: rs \mapsto n)\ @\ \langle c\ ps \to q\rangle &= [q :: ps +\!\!+ rs \mapsto \mathbf{let}\ x = \{q\ (c\ ps)\}\ \mathbf{in}\ n] \\
(r :: rs \mapsto n)\ @\ \langle c\ ps \to q\rangle &= []
\end{aligned}
$$

$$
(y :: rs \mapsto n)\ @\ x = [rs \mapsto \mathbf{let}\ y = x\ \mathbf{in}\ n]
$$

$$
\begin{aligned}
Patterns(D) &= (k\ xs_k)_{k \in D}, && \text{each } xs_k \text{ fresh} \\
Patterns(I) &= (\langle c\ xs_c \to y_c\rangle)_{c \in I}, && \text{each } xs_c, y_c \text{ fresh}
\end{aligned}
\qquad
\begin{aligned}
Patterns(\iota) &= [w], \quad w \text{ fresh} \\
Patterns(\Delta + I\ Rs) &= Patterns(\Delta) +\!\!+ Patterns(I)
\end{aligned}
$$

$$
\begin{aligned}
PatternTypes(D\ Rs, k) &= As, && \text{where } \mathcal{D}(D\ Rs, k) = As \\
PatternTypes(I\ Rs, c) &= \bullet :: As, && \text{where } \mathcal{I}(I\ Rs, c) = As \to B \\
PatternTypes(I\ Rs, c) &= [], && \text{if } \mathcal{I}(I\ Rs, c) \text{ undefined}
\end{aligned}
\qquad
\begin{aligned}
PatternTypes(\iota) &= [] \\
PatternTypes(\Delta + I\ Rs, c) &= \\
&PatternTypes(\Delta, c) +\!\!+ PatternTypes(I\ Rs, c)
\end{aligned}
$$

Figure 5. Pattern Matching Elaboration

The ports are handled left-to-right. The producer is handled first. A different handler for the consumer is invoked depending on whether the producer performs a send command or returns a value.

Notice that the type pollution problem described in Section 2.7 is manifested in the translation to Core Eff: the types of the arguments are unable to distinguish the handled interfaces from those that just happen to be in the ambient ability.

Our pattern matching elaboration procedure is rather direct, but is not at all optimised for efficiency. We believe it should be reasonably straightforward to adapt standard techniques (e.g. [36]) to implement pattern matching more efficiently. However, some care is needed as pattern matching compilation algorithms often reorder columns as an optimisation. Column reordering is not in general a valid optimisation in Frank. This is because commands in the ambient ability, but not in the argument adjustments, are implicitly forwarded, and the order in which they are forwarded is left-to-right. (Precise forwarding behaviour becomes apparent when we combine pattern elaboration with the operational semantics for Core Frank in Section 5.)

4.2 Incomplete and Ambiguous Pattern Matching as Effects

The function PE provides a straightforward means for checking coverage and redundancy of pattern matching. Incomplete coverage can occur iff PE is invoked on three empty lists $PE([], [], [])$, which means PE is undefined on its input. Redundancy occurs iff the final clause defining PE in Figure 5 is invoked in a situation in which us is non-empty. As an extension to Frank, we could allow incomplete and ambiguous pattern matching mediated by effects. We discuss this possibility further in section 9.

5. Small-Step Operational Semantics

We give a small step operational semantics for Core Frank inspired by Kammar et al.'s semantics for the effect handler calculus λ_{eff} [24]. The main differences between their semantics and ours arise from differences in the calculi. Whereas λ_{eff} is call-by-push-value, Core Frank is n-ary call-by-value, which means Core Frank has many more kinds of evaluation context. A more substantive difference is that handlers in λ_{eff} are deep (the continuation reinvokes the handler), whereas handlers in Frank are shallow (the continuation does not reinvoke the handler).

The semantics is given in Figure 6. All of the rules except the ones for handlers are standard β-reductions (modulo some typing noise due to bidirectional typing). We write $n[m/x]$ for n with m substituted for x and $n[\overline{m}/\overline{x}]$ for n with each m_i simultaneously substituted for each x_i. Similarly, we write $n[(n' : P)/f]$ for n with $(n' : P(\overline{R}))$ substituted for $f\ \overline{R}$ and the corresponding generalisation for simultaneous substitution (writing $P(\overline{R})$ for $A[\overline{R}/\overline{Z}]$ where $P = \forall \overline{Z}.A$). Values are handled by substituting the value into the handler's return clause. Commands are handled by capturing the continuation up to the current handler and dispatching to the appropriate clause for the command. We write $HC(\mathcal{E})$ for the set of commands handled by evaluation context $\mathcal{E}$. Formally $HC(-)$ is given by the homomorphic extension of the following equations.

$$
\begin{aligned}
&HC([\ ]) = \varnothing \\
&HC(\mathbf{handle}^{\Delta}_{G}\ \mathcal{E}\ \mathbf{with}\ (c_i\ \overline{x_{c_i}} \to z_{c_i} \mapsto n_{c_i})_i \mid x \mapsto n') = \\
&\qquad\qquad HC(\mathcal{E}) \cup \{c_i\}_i
\end{aligned}
$$

The side condition on the command rule ensures that command c' is handled by the nearest enclosing handler that has a clause for

$$
\begin{array}{rl}
\text{(use values)} & v ::= x \mid f\ \overline{R} \mid c \mid (w : A) \\
\text{(construction values)} & w ::= v \mid k\ \overline{w} \mid \lambda\overline{x}.n \\
\text{(evaluation contexts)} & \mathcal{E} ::= [\ ] \mid \mathcal{E}\ \overline{n} \mid v\ (\overline{w}, \mathcal{E}, \overline{n}) \mid (\mathcal{E} : A) \mid k\ (\overline{w}, \mathcal{E}, \overline{n}) \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ (k\,\overline{x_k} \mapsto n_k)_k \\
& \mid\ \mathbf{handle}^{\Delta}_{G}\ \mathcal{E}\ \mathbf{with}\ (c\,\overline{x_c} \to z_c \mapsto n_c)_c \mid x \mapsto n' \\
& \mid\ \mathbf{let}\ (f : P) = \mathcal{E}\ \mathbf{in}\ n
\end{array}
$$

$$
\begin{array}{rcl}
(\lambda\overline{x}.n : \{\overline{A \to}\ [\Sigma]B\})\ \overline{w} & \longrightarrow & (n[\overline{(w : A)}/\overline{x}] : B) \\
\mathbf{case}\ (k'\ \overline{w} : D\,\overline{R})\ \mathbf{of}\ (k\,\overline{x_k} \mapsto n_k)_k & \longrightarrow & n_{k'}[\overline{(w : A)}/\overline{x_{k'}}], \quad \overline{A} = \mathcal{D}(D\,\overline{R}, k') \\
\mathbf{handle}^{\Delta}_{G}\ v\ \mathbf{with}\ (c\,\overline{x_c} \to z_c \mapsto n_c)_c \mid x \mapsto n' & \longrightarrow & n'[v/x] \\
\mathbf{handle}^{\Delta}_{G}\ \mathcal{E}[c'\ \overline{w}]\ \mathbf{with}\ (c\,\overline{x_c} \to z_c \mapsto n_c)_c \mid x \mapsto n' & \longrightarrow & n_{c'}[\overline{(w : A)}/\overline{x_{c'}}, (\lambda y.\mathcal{E}[y] : \{B \to G\})/z_{c'}], \quad c' \notin HC(\mathcal{E}) \text{ and } c' : \overline{A \to}\ B \in \Delta \\
\mathbf{let}\ f : P = w\ \mathbf{in}\ n & \longrightarrow & n[\overline{(w : P)}/f] \\
\mathbf{letrec}\ \overline{f : P = \lambda\overline{x}.n}\ \mathbf{in}\ n' & \longrightarrow & n'[\overline{(\lambda\overline{x}.\mathbf{letrec}\ \overline{f : P = \lambda\overline{x}.n}\ \mathbf{in}\ n : P)/f}]
\end{array}
$$

$$
(v : A) \longrightarrow v
$$

$$
\mathcal{E}[n] \longrightarrow \mathcal{E}[n'], \quad \text{if } n \longrightarrow n'
$$

Figure 6. Small-Step Operational Semantics for Core Frank

handling c'. A more intensional way to achieve the same behaviour is to explicitly forward unhandled commands using an additional rule [24].

Reduction preserves typing.

THEOREM 1 (Subject Reduction).

- *If* $\Gamma\ [\Sigma]\vdash m \Rightarrow A$ *and* $m \longrightarrow m'$ *then* $\Gamma\ [\Sigma]\vdash m' \Rightarrow A$.
- *If* $\Gamma\ [\Sigma]\vdash n : A$ *and* $n \longrightarrow n'$ *then* $\Gamma\ [\Sigma]\vdash n' : A$.

There are two ways in which reduction can stop: it may yield a value, or it may encounter an unhandled command instance (if the ambient ability is non-empty). We capture both possibilities with a notion of normal form, which we use to define type soundness.

DEFINITION 2 (Normal Forms). *If* $\Gamma\ [\Sigma]\vdash n : A$ *then we say that* n *is normal with respect to* Σ *if it is either a value* w *or of the form* $\mathcal{E}[c\ \overline{w}]$ *where* $c : \overline{A} \to B \in \Sigma$ *and* $c \notin HC(\mathcal{E})$.

THEOREM 3 (Type Soundness).
If $\cdot\ [\Sigma]\vdash n : A$ *then either* n *is normal with respect to* Σ *or there exists* $\cdot\ [\Sigma]\vdash n' : A$ *such that* $n \longrightarrow n'$. *(In particular, if* $\Sigma = \varnothing$ *then either* n *is a value or there exists* $\cdot\ [\Sigma]\vdash n' : A$ *such that* $n \longrightarrow n'$.*)*

6. 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 [53].

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

As discussed in Section 3, the general rule is that if the body of a data type (or interface) definition includes the implicit effect variable then the first parameter of the definition is also the implicit effect variable. Initially a log contains a parser computation (`start`). 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 [LookAhead, Abort, Console] X ->
         Buffer ->
         <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>     =
  on inch!
     { '\b' -> rollback l
     | c    -> input (inched l k) (hold c) (k c) }
input l _        <aborting -> k> = rollback l
```

Note that the `Log` type's ability has been instantiated with exactly the same ambient ability as is offered at the port in which the parser plugs. 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 [LookAhead, Abort, Console] X ->
           [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, Console]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.

While the `Log` type does what is required of it, this example does expose a shortcoming of Frank as currently specified: we have no means to prevent the parser process from accessing `Console` commands, because our adjustments can add and shadow interfaces but not remove them. If we permitted 'negative' adjustments, we could give the preferable types

```
input :  Log [LookAhead, Abort] X -> Buffer ->
         <LookAhead, Abort, -Console>X ->
         [Console]X

rollback : Log [LookAhead, Abort] X -> [Console]X

parse : {[LookAhead, Abort]X} -> [Console]X
```

At time of writing, it is clear how to make negative adjustments act on a concrete ability, but less clear what their impact is on effect polymorphism—a topic of active investigation.

7. Implementation

The second author has been plotting Frank since at least 2007 [37]. In 2012, he implemented a prototype for a previous version of Frank [39]. 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 now implemented a prototype of Frank in Haskell that matches the design described in the current paper [33]. 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 and Ağacan [2], the "type-inference-in-context" technique of Gundry et al. [16], and the existing implementation of Shonky [41].

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 and Ağacan [2] introduce an extension to parsing frameworks based on parsing expression grammars. Such grammars provide a formal basis for the Parsec [30] and Trifecta [27] parser combinator Haskell libraries. In contrast to the ad hoc methods typically employed by many indentation sensitive languages (including Haskell and Idris [8]), Adams and 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 [16] for implementing type inference and unification (Gundry's thesis [17] contains a more detailed and up-to-date account). The key insight is to keep track in the context not just of term variables but also 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 [17].

The back-end of the prototype diverges from the formalism described in the paper. Instead of targeting a core calculus, Frank is translated into Shonky [41], which amounts to an untyped version of Frank. Shonky executes code directly through an abstract machine much like that of Hillerström and Lindley [21].

8. Related Work

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

Efficient Effect Handler Implementations A natural implementation for handlers is to use *free monads* [24]. Swierstra [55] 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 [24].

Wu and Schrijvers [62] show how to obtain a particularly efficient implementation of deep handlers taking advantage of fusion. Their work explains how Kammar et al. [24] efficient handler code in Haskell. Kiselyov and Ishii [25] optimise their shallow effect handlers implementation, which is based on free monads, by taking advantage of an efficient representation of sequences of monadic operations [58]. The experimental multicore extension to OCaml [11] 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 [29] 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 [13] and layered monads [12] 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. [54] 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. [52] compare the expressiveness of effect handlers and monad transformers in the context of Haskell. Forster et al. [14] compare effect handlers with monadic reflection and delimited control in a more abstract setting.

Variations and Applications Lindley [32] investigates an adaptation of effect handlers to more restrictive forms of computation based on idioms [42] and arrows [23]. Wu et al. [63] 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 and Ishii [25] provide solutions to particular instances of this problem. Schrijvers et al. [51] apply effect handlers to logic programming.

9. Future Work

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

Direct Semantics The semantics of Frank presented in this paper is via a translation to Core Frank, a fairly standard call-by-value language extended with algebraic effects and unary effect handlers. This has the advantage of making it clear how the semantics of Frank relates to the semantics of other languages with algebraic effects and effect handlers. Nevertheless, we believe there are good reasons to explore a more direct semantics. A direct semantics may offer a more efficient implementation or more parsimonious generated code (witness the translation of `pipe` into Core Frank in Section 4). Perhaps a more compelling motivation is that multihandlers appear to share some features of other expressive programming abstractions such as multiple dispatch and join patterns [15], 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 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 lists as follows:

```
interface ListI X = nil : 0
                  | cons : X -> ListI X -> 0
```

The type of lists with elements of type `X` is `[ListI X]0`. We can then simulate `map` as follows:

```
mapI : {X -> Y} -> [ListI X]0 -> [ListI Y]0
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 algebraic account of type checking and elaboration [3] 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 As discussed in Section 4.2 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, Schrijvers and Hinze [63] 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.

Scaling by Naming What if we want to have multiple `State` components? One approach, adopted by Brady [7], is to rename them apart: when we declare the `State` interface, we acquire also the `Foo.State` interface with operations `Foo.get` and `Foo.set`, for any `Foo`. We would then need to specialise `State`ful operators to a given `Foo`, and perhaps to generate fresh `Foo`s dynamically.

Dynamic Effects An important effect that we cannot implement directly in Frank as it stands is dynamic allocation of ML-style references. One difficulty is that the `new` command which allocates a new reference cell has a polymorphic type `forall X.new : Ref X`. But even if we restrict ourselves to a single type, it is still unclear how to safely represent the `Ref` data type. Eff works around the problem using a special notion of resource [6]. We would like to explore adding resources or a similar abstraction to Frank.

Negative Adjustments Currently the only non-trivial adjustments are positive: they add interfaces to an ability. As mentioned in Section 6, we would like to add support for negative adjustments that remove interfaces from an ability. An extreme case of a negative adjustment is a *nugatory* adjustment that removes all interfaces from any ability to yield the pure ability. Whereas positive adjustments can be simulated with more conventional effect type systems, at the cost of some precision in types, it is not clear to us whether negative adjustments can be.

Controlled Snooping We might consider allowing handlers to trap some or even all commands generically, just as long as their ports 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 [38] observes that indexing input and output types over the state effectively lets us specify interfaces in a proof-relevant Hoare logic. Hancock and Hyvernat [18] 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 [35] and Haskell type classes [61].

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', whilst 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 [1]. 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 [4].

Given that we have the means to negotiate purity locally whilst 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 [40].

10. 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, 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.

References

[1] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In Morrisett and Uustalu [43], pages 185–196.

[2] M. D. Adams and O. S. Ağacan. Indentation-sensitive parsing for Parsec. In *Haskell*. ACM, 2014.

[3] R. Atkey. An algebraic approach to typechecking and elaboration, 2015. URL `http://bentnib.org/docs/algebraic-typechecking-20150218.pdf`.

[4] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In Morrisett and Uustalu [43], pages 197–208.

[5] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014.

[6] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[7] E. Brady. Programming and reasoning with algebraic effects and dependent types. In Morrisett and Uustalu [43], pages 133–144.

[8] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[9] M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors. *ICFP*, 2011. ACM.

[10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

[11] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, 2015.

[12] A. Filinski. Representing layered monads. In A. W. Appel and A. Aiken, editors, *POPL*, pages 175–188. ACM, 1999.

[13] A. Filinski. Monads in action. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 483–494. ACM, 2010.

[14] Y. Forster, O. Kammar, S. Lindley, and M. Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *CoRR*, abs/1610.09161, 2012.

[15] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In H. Boehm and G. L. S. Jr., editors, *POPL*, pages 372–385. ACM, 1996.

[16] A. Gundry, C. McBride, and J. McKinna. Type inference in context. In *MSFP*. ACM, 2010.

[17] A. M. Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[18] P. Hancock and P. Hyvernat. Programming interfaces and basic topology. *Ann. Pure Appl. Logic*, 137(1-3):189–239, 2006.

[19] D. Hillerström. Handlers for algebraic effects in Links. Master's thesis, School of Informatics, The University of Edinburgh, 2015.

[20] D. Hillerström. Compilation of effect handlers and their applications in concurrency. Master's thesis, School of Informatics, The University of Edinburgh, 2016.

[21] D. Hillerström and S. Lindley. Liberating effects with rows and handlers. In J. Chapman and W. Swierstra, editors, *TyDe*, pages 15–27. ACM, 2016.

[22] R. Hinze and J. Voigtländer, editors. *MPC*, volume 9129 of *Lecture Notes in Computer Science*, 2015. Springer.

[23] J. Hughes. Programming with arrows. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 73–129. Springer, 2004.

[24] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In Morrisett and Uustalu [43], pages 145–158.

[25] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In B. Lippmeier, editor, *Haskell*, pages 94–105. ACM, 2015.

[26] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In C. Shan, editor, *Haskell*, pages 59–70. ACM, 2013.

[27] E. A. Kmett. Trifecta (1.5.2), 2015. `http://hackage.haskell.org/package/trifecta-1.5.2`.

[28] D. Leijen. Koka: Programming with row polymorphic effect types. In P. Levy and N. Krishnaswami, editors, *MSFP*, volume 153 of *EPTCS*, pages 100–126, 2014.

[29] D. Leijen. Type directed compilation of row-typed algebraic effects. In A. D. Gordon, editor, *POPL*. ACM, 2017.

[30] D. Leijen and P. Martini. Parsec (3.1.9), 2015. `http://hackage.haskell.org/package/parsec-3.1.9`.

[31] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

[32] S. Lindley. Algebraic effects and effect handlers for idioms and arrows. In J. P. Magalhães and T. Rompf, editors, *WGP*, pages 47–58. ACM, 2014.

[33] S. Lindley, C. McBride, and C. McLaughlin. Frank junior, 2016. `https://www.github.com/cmcl/frankjnr`.

[34] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM, 1988.

[35] D. B. MacQueen. Modules for standard ML. In *LISP and Functional Programming*, pages 198–207, 1984.

[36] L. Maranget. Compiling pattern matching to good decision trees. In *ML*, pages 35–46. ACM, 2008.

[37] C. McBride. How might effectful programs look? In *Workshop on Effects and Type Theory*, 2007. `http://cs.ioc.ee/efftt/mcbride-slides.pdf`.

[38] C. McBride. Kleisli arrows of outrageous fortune, 2011. Draft. `https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf`.

[39] C. McBride. Frank (0.3), 2012. `http://hackage.haskell.org/package/Frank`.

[40] C. McBride. Turing-completeness totally free. In Hinze and Voigtländer [22], pages 257–275.

[41] C. McBride. Shonky, 2016. `https://github.com/pigworker/shonky`.

[42] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[43] G. Morrisett and T. Uustalu, editors. *ICFP*, 2013. ACM.

[44] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[45] G. D. Plotkin and J. Power. Semantics for algebraic operations. *Electr. Notes Theor. Comput. Sci.*, 45:332–345, 2001.

[46] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In F. Honsell and M. Miculan, editors, *FOSSACS*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[47] G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *FOSSACS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[48] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categ. Structures*, 11(1):69–94, 2003.

[49] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[50] M. Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014.

[51] T. Schrijvers, N. Wu, B. Desouter, and B. Demoen. Heuristics entwined with handlers combined: From functional specification to logic programming implementation. In O. Chitil, A. King, and O. Danvy, editors, *PPDP*, pages 259–270. ACM, 2014.

[52] T. Schrijvers, M. Pirò̀g, N. Wu, and M. Jaskelioff. Monad transformers and modular algebraic effects. Technical report, University of Leuven, 2016.

[53] O. Shivers and A. J. Turon. Modular rollback through control logging: a pair of twin functional pearls. In Chakravarty et al. [9], pages 58–68.

[54] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In Chakravarty et al. [9], pages 15–27.

[55] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[56] W. Swierstra, editor. *Haskell*, 2014. ACM.

[57] J. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[58] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In Swierstra [56], pages 133–144.

[59] K. Vonnegut. *Deadeye Dick*. Delacorte Press, 1982.

[60] P. Wadler. The essence of functional programming. In R. Sethi, editor, *POPL*, pages 1–14. ACM, 1992.

[61] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM, 1989.

[62] N. Wu and T. Schrijvers. Fusion for free - efficient algebraic effect handlers. In Hinze and Voigtländer [22], pages 302–322.

[63] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In Swierstra [56], pages 1–12.