Do Be Do Be Do

Sam Lindley
The University of Edinburgh

Conor McBride
University of Strathclyde

Craig McLaughlin
The University of Edinburgh

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 the special case of a Frank *operator* that interprets no commands. Moreover, Frank's operators can be *multihandlers* which simultaneously interpret commands from several sources at once, without disturbing the direct style of functional programming with values.

Effect typing in Frank employs a novel form of effect polymorphism which avoids all mention of 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 multihandlers 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.

1. Introduction

Shall I be pure or impure?

—Philip Wadler

We say 'Yes.': this is a choice to make *locally*. We introduce **Frank**, an applicative language where the ambient ability to express 'impure' computations is open to renegotiation, based on Plotkin and Power's algebraic effects [30–33], in conjunction with Plotkin and Pretnar's handlers for algebraic effects [34, 35]—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.

Frank programs are written in direct style in the spirit of effect type systems [22, 43]. Frank *operators* generalise functions to handle effects. An effect handler acts as an interpreter for a specified set of commands whose interfaces are statically tracked by the type system. A function is the special case of an operator whose handled command set is empty.

The contributions of this paper are:

- Frank, a strict functional programming language featuring a bidirectional effect type system, effect polymorphism, and effect handlers;
- a novel approach to effect polymorphism which avoids all mention of effect variables, crucially relying on the observation that one must always instantiate the effects of a function being applied with the current *ambient ability*;
- operators as both *multihandlers* for handling multiple computations over distinct effect sets simultaneously and as *functions* acting on values;
- a description of pattern matching compilation from Frank into a core language, *Core Frank*;
- a straightforward small-step operational semantics for Core Frank and a proof of type soundness;
- an exploration of future research, combining effect-and-handlers programming with substructural typing, dependent types and totality.

Whilst we have implemented and learned from various prototypes of Frank, we do not yet have a full implementation matching the current system described in the paper.

A number of other languages and libraries are built around effect handlers and algebraic effects. Bauer and Pretnar's Eff [4] 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 [3]. Their implementation [36] supports Hindley-Milner type inference, and the type system incorporates effect sub-typing. In contrast, Frank uses bidirectional type inference, and avoids sub-typing altogether.

Kammar et al [15] describe a number of effect handler libraries for languages ranging from Racket, to SML, to Haskell. Apart from the Haskell library, these other 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. Furthermore, although there are a number of ways of almost simulating effect type systems in Haskell, none is without its flaws. Kiselyov and collaborators [16, 17] have designed another

Haskell library for effect handlers, making a different collection of design choices.

Brady [5] has designed a library and DSL for programming with effects in his dependently typed Idris language. Like the Haskell libraries, Brady's library currently requires the programmer to write effectful code in a monadic style.

The rest of the paper is structured as follows. Section 2 introduces Frank by example. Section 3 presents a type system for Frank. Section 4 describes how to elaborate multihandlers and pattern matching into Core Frank, a language of plain call-by-value functions, explicit case analysis and unary handler constructs. Section 5 gives a semantics for Core Frank and proves types soundness. Section 6 discusses the means to store computations in data structures. Section 7 outlines related work and Section 8 discusses future work.

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

Frank is a functional programming language with effects and handlers in the style of Eff [4] controlled by a type system inspired by Levy's call-by-push-value [19]. Doing and Being are clearly separated, and managed by distinguished notions of computation and value types.

2.1 Datatypes and First-Order Functions

Concrete values live in inductive datatypes. By convention (not compulsion), type constructors get uppercase initials, and may apply prefixed to parameters, also written uppercase. Data constructors are prefix and, 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
```

Note that `cons` is not a function and thus cannot be 'partially 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 Computations in Ambient Silence

To do is to be: 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)
```

So, a value type `A` is a datatype `D A1 ... An`, a suspended computation type `{C}`, or a type variable `X`. A computation type resembles a function type `T1 -> ... -> Tn -> [I1 ... In]B` with n *ports* and a *peg* showing the *ability* the computation needs—a bracketed list of enabled *interfaces*—and the *value type* it delivers. In Frank, names stand for values (a simplifying decision which we shall re-examine in section 8). 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
```

but have a care: the empty bracket stands for the *ambient* ability, not for pure inability.

The type of `map` in Frank says that whatever ability a usage 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 something like its monadic `mapM`, as we shall see just as soon as we acquire nontrivial ability.

2.3 Controlling Evaluation

Frank is a call-by-value language, so we should be careful when defining control operators. E.g., we may define a sequential composition operators

```
fst : X -> Y -> X
fst x y = x

snd : X -> Y -> Y
snd x y = y
```

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

Meanwhile, if we want to avoid evaluation, we must do so explicitly. This

```
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 suspend:

```
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, what they *are* is suspended computations of type `{X}`, but we must *do* one. To be is to do: the postfix `!` denotes nullary application of a suspended computation. (Other notational possibilities exist, but we defer their discussion to section 8.)

Suspended computations are written 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, e.g.,

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

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

Non-nullary suspensions let us simulate case-expressions inline by 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 are collections of parameterised interfaces, each of which describes a choice of *commands* (also known as *operations*, elsewhere in the literature). 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
                     | set : 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 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.

We may use our silent ability polymorphism nontrivially, to make `map` `send` a list of things, 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 requested (or threatened, if you like) by the return type of `bad`.

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

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

While we retain Milner's insight that *value type* polymorphism can and should be specialised by means of unification, we do not imagine that all implicit parameters must be found by unification. Frank's *ability* polymorphism is specialised simply by substituting for the formal ambient ability of a given operation the actual ambient ability at each point of its use. We do not imagine that Frank can deliver all the effect polymorphism a cunning programmer could ever want, but we do aim to show that you can achieve quite a lot of power without, e.g., solving constraints on row variables. Spring the 'parametric polymorphism means unification' trap!

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. E.g., we can implement the result of reading a list of lists until one is empty and concatenating the result.

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

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

```
catter = do                -- Haskell
  xs <- receive
  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 whose only purpose is to tell the compiler where to parse a type as $\mathit{effect\ value}$ 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`. The choice to give names only to values offers conceptual simplicity but forces us to pay a `!` for nullary computations seldom passed as values. We revisit this dilemma in section 8.

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

```
next : [State Nat]Nat
next! = fst get! (put (suc get!))
```

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.

Contrastingly, an *explicit environment* [37] coroutines with the expression it governs by repeated interaction, supplying definitions of the packaged fields on demand, but forwarding the eventual value with no post-processing. An exception handler is another means for one process to contextualise the execution of another, remaining entirely dormant and dismounting itself without fuss if the subordinate runs without failure, but stepping into the breach if needs be.

Frank is already distinct from Eff in its 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, we use the blank space application notation for more general modes of coroutining between operator and arguments than the return-value-then-quit default. E.g., 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 _ <set 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 Nat X)
index xs = state zero (map {x -> pair next! x} xs)
```

If you will allow string notation for lists of characters and decimal numerals, we obtain:

```
index "abc" = cons (pair 0 'a')
              (cons (pair 1 'b')
                (cons (pair 2 'c') nil))
```

What is happening?

Firstly, the type of `state` shows us that Frank operations 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 empty as heretofore. 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 Nat`, allowing the elementwise operation to invoke `next!`.

Secondly, having offered the `State S` interface, the `state` operator must detect it. Its first equation 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. Requests are matched by patterns in chevrons which show the particular command being handled left of `->`, with a pattern variable standing for the *continuation* on the right. The patterns of `state` thus detects all the signals advertised as acceptable at its ports.

Thirdly, having detected the signals, the `state` operator should *handle* them. In the first equation, the returned `x` is forwarded forthwith. In the case of a command, we *reinvoke* `state`, with the new state (the old state for `get` and the given state for `put s`) in the first port and the continuation *invoked* in the second port. We emphasise a key difference from Eff: Frank's continuation variables are 'shallow'—they capture only the rest of the subordinated computation, not the result of handling it, allowing us to change how we carry on handling, e.g., by updating the state. The 'deep' approach forces a higher-order implementation, interpreting a stateful computation as a function from the initial state. We can simulate shallow by deep, as follows, inserting the recursive calls to `state'` that Eff would bundle inside `k` already.

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

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. By contrast, 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.

2.7 Handling on Multiple Ports

We can write n-ary operators, so we can offer different adjustments to the ambient ability at different ports. E.g., 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 the `pipe` operator 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 the `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: this is the case where the consumer has delivered a value, so we may safely kill the producer. 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 we can only `abort` with a 'broken pipe'.

We can run this 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. The other way around also works as it should.

```
pipe (pipe
      (sends (cons "do" (cons "be" (cons "" nil))))
      spacer!) catter!
  = "do be "
```

Again, 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.

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 have `Abort`. However, because neither port advertises `Abort` in its adjustment, the implementation of `pipe` may not intercept the `aborting` command. In particular, the `<_>` pattern matches only the behaviours 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 signals. Of course, the `pipe` operation might prevent action by ignoring the continuation to a `send` on its first port or a `receive` on its second, but it will not change the meaning of other things which do happen.

2.8 The Catch Question

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

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

The catch alternative is given as a suspended computation allowing us to choose whether to run it. We do not presume that the ambient ability in which `catch` executes itself offers the `Abort` interface, where the typical treatment of exceptions treat `catch` as the prioritised choice between two failure-prone computations. E.g., the Haskell library offers

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

unnecessarily making the ability to abort non-local. Leijen deserves credit for making a similar observation in Koka's treatment of exceptions [18].

Frank's ability 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`.

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 Eff-style computation-to-computation handler types distinct from value-to-computation function types [3].

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 minimally. We conceal the machinery by which commands from the input not handled locally must be forwarded with suitably wrapped continuations. Of course, that is exactly what happens, 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. Static Semantics

A value is. A computation does.

—Paul Blain Levy

In this section we give a formal presentation of Frank's type system.

3.1 Syntax

The abstract syntax of Frank is given in Figure 1.

Notation The ? superscript denotes that its subject is optional and an overbar denotes a list of zero or more elements. For instance, in $D\ \Sigma^?\ \overline{A}$ the parameterised data type D optionally takes an ability $\Sigma^?$ and also takes a list of zero or more value types $\overline{A}$.

The types are divided into value types and computation types. Value types are pure data types ($D\ \overline{A}$), effect-parametric data types ($D\ \Sigma\ \overline{A}$), suspended computation types ($\{C\}$), or type variables (X).

Types

$$
\begin{array}{lll}
\text{(value types)} & A, B & ::= D\ \Sigma^?\ \overline{A} \mid \{C\} \mid X \\
\text{(computation types)} & C & ::= \overline{T \to G} \\
\text{(ports)} & T & ::= \langle \Delta \rangle A \\
\text{(pegs)} & G & ::= [\Sigma] A \\
\text{(type/effect variables)} & Z & ::= X \mid \varepsilon \\
\text{(polytypes)} & P & ::= \forall \overline{Z}.A \\
\text{(abilities)} & \Sigma & ::= \varnothing \mid \Sigma, I\ \overline{A} \mid \varepsilon \\
\text{(adjustments)} & \Delta & ::= \iota \mid \Delta + I\ \overline{A} \\
\text{(type environments)} & \Gamma & ::= \cdot \mid \Gamma, x : A \mid f : P
\end{array}
$$

Terms

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

Figure 1. Frank Abstract Syntax

Computations types are build from input *ports* T and output *pegs* G. A computation type represents a multihandler. An n handler has type

$$\langle \Delta_1 \rangle A_1 \to_1 \ \cdots \to \langle \Delta_n \rangle A_n \to [\Sigma] B$$

For each i, the multihandler 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:

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

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 the rather more verbose:

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

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

$$\frac{A \equiv \langle \iota \rangle A}{I_1\ \overline{A_1}, \ldots, I_n\ \overline{A_n} \equiv \iota + I_1\ \overline{A_1} + \cdots + I_n\ \overline{A_n}}$$

Similarly, we adopt the convention that the effect variable ε may be omitted from abilities and ports.

$$\frac{A \equiv [\varepsilon] A}{I_1\ \overline{A_1}, \ldots, I_n\ \overline{A_n} \equiv \varepsilon, I_1\ \overline{A_1}, \ldots, I_n\ \overline{A_n}}$$

Given this syntactic sugar we need never write the effect variable ε anywhere in a Frank program.

We let X range over type variables and ε range over effect variables; polytypes may be polymorphic in both. Though we need

never write 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 $\mathcal{D}(D\ \Sigma^?\ \overline{A}, k)$ for the signature of constructor k of data type $D\ \Sigma^?\ \overline{A}$. A definition for effect interface $I(\overline{X})$ consists of a collection of command declarations 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 $\overline{X}$. Each command belongs to a single interface and may appear only once in that interface. We write $\mathcal{I}(I\ \overline{A}, c)$ for the signature of command c of effect interface $I\ \overline{A}$

An ability is a collection of interfaces initiated either with the empty ability $\varnothing$ (yielding a *closed* ability) or an effect variable ε (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.

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

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

Type environments distinguish monomorphic and polymorphic variables.

Frank follows a bidirectional typing discipline [29]. 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 operator 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 an empty spine.

A computation is defined by a sequence of pattern matching clauses ($\overline{r \mapsto n}$). Each pattern matching clause takes a sequence of computation patterns ($\overline{r}$). A computation pattern is either a standard value pattern (p), a command pattern ($\langle c\ \overline{p} \to g\rangle$), which matches command c binding its arguments to $\overline{p}$ and the continuation to g, or a thunk pattern $\langle x\rangle$, which matches any 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 from Section 2.

$$\begin{array}{l}
\mathbf{letrec}\ \mathsf{map} : \\
\quad \forall \varepsilon\ X\ Y.\{\langle\iota\rangle\{\langle\iota\rangle X \to [\varepsilon]Y\} \to \langle\iota\rangle(\mathsf{List}\ X) \to [\varepsilon](\mathsf{List}\ Y)\} \\
\quad\quad = f\ \mathsf{nil} \qquad\quad \mapsto \mathsf{nil} \\
\quad\quad\quad f\ (\mathsf{cons}\ x\ \mathsf{xs}) \mapsto \mathsf{cons}\ (f\ x)\ (\mathsf{map}\ f\ \mathsf{xs})\ \mathbf{in} \\
\mathbf{letrec}\ \mathsf{state} : \forall \varepsilon\ X.\{\langle\iota\rangle X \to \langle \iota + \mathsf{State}\ S\rangle X \to [\varepsilon]X\} \\
\quad\quad = z_0\ x \qquad\quad \mapsto x \\
\quad\quad\quad s\ \ \langle \mathsf{get} \mapsto g\rangle \mapsto \mathsf{state}\ s\ (g\ s) \\
\quad\quad\quad z_1\ \langle \mathsf{set} \mapsto g\rangle \mapsto \mathsf{state}\ s\ (g\ unit)\ \mathbf{in} \\
\mathbf{let}\ \mathsf{index} : \forall \varepsilon\ X.\{\langle\iota\rangle \mathsf{List}\ X \to [\varepsilon]\mathsf{List}\ (\mathsf{Pair\ Nat}\ X)\} = \\
\quad\quad = \{xs \mapsto \mathsf{state\ zero}\ (\mathsf{map}\ \{x \mapsto \mathsf{pair\ next!}\ x\}\ \mathsf{xs})\}\ \mathbf{in} \\
\mathsf{index}\ \text{“abc”}
\end{array}$$

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 translated to explicit pattern matching in the obvious 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 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 judgement $\Gamma\ [\Sigma]\vdash\ n : A$ states that in type environment Γ with ambient ability Σ, construction n has type A. The judgement $\Gamma \vdash e : C$ states that in type environment Γ, computation e has type C. The judgement $r : G \dashv [\Sigma]\ \Gamma$ states that computation pattern r of peg type G with ambient ability Σ binds type environment Γ. The judgement $p : A \dashv [\Sigma]\ \Gamma$ states that value pattern p of type A with ambient ability Σ binds type environment Γ. The ambient ability is only required in the latter case in order to instantiate effect polymorphic data types with the ambient ability.

We infer the type of a monomorphic variable (x) by looking it up in the environment. We do the same for a polymorphic variables (f), but also instantiate its type variables, and in particular instantiate effect variables with the ambient ability. The type of a command (c) is inferred by looking it up in the ambient ability, where the ports have the identity adjustment and the peg has the ambient ability.

To infer the type of an operator application $m\ \overline{n}$ under ambient ability Σ we first infer the type of the multihandler m of the form $\{\overline{\langle\Delta\rangle A \to [\Sigma]B}\}$. We then check that each argument n_i matches the inferred type in the ambient ability extended with adjustment Δ_i. If this succeeds, then the inferred type for the operator application is B.

Any use (m) is also a construction (but not vice-versa). To check a data type ($k\ \overline{v}$), thunk ($\{e\}$) polymorphic let, $\mathbf{let}\ f : P = n\ \mathbf{in}\ n'$ or mutual recursion $\mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n$) we 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 $\overline{r} \mapsto n$, where $\overline{r}$ is a sequence of computation patterns whose variables are bound in n. The side condition on the computation introduction 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. A command pattern $\langle c\ \overline{p} \to g\rangle$ may be checked at type $\langle\Delta\rangle B'$ with ambient ability Σ. 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.

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

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

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

$$\frac{x : A \in \Gamma}{\Gamma\,[\Sigma]\vdash x \Rightarrow A} \qquad \frac{f : \forall \overline{Z}.A \in \Gamma \qquad \text{if } \varepsilon \in dom(\theta) \text{ then } \theta(\varepsilon) = \Sigma}{\Gamma\,[\Sigma]\vdash f \Rightarrow \theta(A)} \qquad \frac{c : \overline{A} \to B \in \Sigma}{\Gamma\,[\Sigma]\vdash c \Rightarrow \{\overline{\langle \iota \rangle A \to}\,[\Sigma]B\}}$$

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

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

$$\frac{\Gamma\,[\Sigma]\vdash m \Rightarrow A \qquad A = B}{\Gamma\,[\Sigma]\vdash m : B} \qquad \frac{\Gamma \vdash e : C}{\Gamma\,[\Sigma]\vdash \{e\} : \{C\}}$$

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

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

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

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

$$\frac{p : A \dashv [\Sigma]\ \Gamma}{p : \langle \Delta \rangle A \dashv [\Sigma]\ \Gamma} \qquad \frac{c : \overline{A} \to B \in \varnothing \oplus \Delta \qquad (p_i : A_i \dashv [\Sigma]\ \Gamma_i)_i}{\langle c\,\overline{p} \to g \rangle : \langle \Delta \rangle B' \dashv [\Sigma]\ \overline{\Gamma}, g : \langle \iota \rangle B \to [\Sigma \oplus \Delta]B'} \qquad \overline{\langle x \rangle : \langle \Delta \rangle A \dashv [\Sigma]\ x : \{[\Sigma \oplus \Delta]A\}}$$

$\boxed{p : A \dashv [\Sigma]\ \Gamma}$

$$\overline{x : A \dashv [\Sigma]\ x : A} \qquad \frac{k\,\overline{A} \in D\,\Sigma^?\,\overline{B} \qquad \overline{p : A \dashv [\Sigma]\ \Gamma}}{k\,\overline{p} : D\,\Sigma^?\,\overline{B} \dashv [\Sigma]\ \overline{\Gamma}}$$

Figure 2. Frank Typing Rules

Note that the syntactic sugar $\mathbf{let}\ x = n\ \mathbf{in}\ n$ differs from the built-in $\mathbf{let}\ x = (n : P)\ \mathbf{in}\ n'$ construct in two ways: 1) it does not include the type annotation P on n, and 2) x is monomorphic in n'.

4. Core Frank

We elaborate Frank into Core Frank, a language in which multihandlers are replaced by a combination of call-by-value functions, case statements, and unary effect handlers. Multihandlers in Frank are elaborated to n-ary functions over suspended computations in Core Frank. Shallow pattern matching on a single computation is elaborated to unary effect handling. Shallow pattern matching on a data type value is elaborated to case analysis. Deep pattern matching on multiple computations is elaborated 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. We allow constructions to be treated as uses by supplying a polytype annotation. This is helpful for defining a small step-operational semantics (Section 5) as we sometimes need to substitute a let-bound polymorphic construction for a use. 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 command and thunk 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. Instead of general operator application, we have only 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. In the handler introduction rule, the adjustment Δ is needed by the premises. In the source language this adjustment is built into pegs. In the rule for coercing a polytype annotated construction to a use, we must instantiate the polymorphism as the inference judgement yields a plain value type.

Notation We extend our indexed list notation to indexing over data constructors and commands. In typing rules, we follow the convention that if a metavariable only appears 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.

Types

$$
\begin{array}{lll}
\text{(value types)} & A, B ::= & D\ \overline{A} \mid \{C\} \mid X \\
\text{(computation types)} & C ::= & \overline{A} \to G \\
\text{(pegs)} & G ::= & [\Sigma]A \\
\text{(type/effect variables)} & Z ::= & X \mid \varepsilon \\
\text{(polytypes)} & P ::= & \forall \overline{Z}.A \\
\text{(abilities)} & \Sigma ::= & \varnothing \mid \Sigma, I\ \overline{A} \mid \varepsilon \\
\text{(adjustments)} & \Delta ::= & \iota \mid \Delta + I\ \overline{A} \\
\text{(type environments)} & \Gamma ::= & \cdot \mid \Gamma, x : A \mid f : P
\end{array}
$$

Terms

$$
\begin{array}{lll}
\text{(uses)} & m ::= & x \mid f \mid c \mid m\ s \mid (n : P) \\
\text{(constructions)} & n ::= & m \mid k\ \overline{n} \\
& \mid & \lambda \overline{x}.n \\
& \mid & \mathbf{case}\ m\ \mathbf{of}\ \overline{k\ \overline{x} \mapsto n} \\
& \mid & \mathbf{handle}^{\Delta}\ m\ \mathbf{with}\ \overline{c\ \overline{x} \to g \mapsto n}\ \overline{\mid\ x \mapsto n'} \\
& \mid & \mathbf{let}\ f : P = n\ \mathbf{in}\ n' \\
& \mid & \mathbf{letrec}\ \overline{f : P = \lambda \overline{x}.n}\ \mathbf{in}\ n' \\
\text{(spines)} & s ::= & \overline{n}
\end{array}
$$

Figure 3. Core Frank Abstract Syntax

4.1 Elaboration

We now describe how to elaborate Frank into Core Frank by way of a translation $[\![-]\!]$. The translation on types is given simply by the homomorphic extension of the following equation on ports

$$[\![\langle \Delta \rangle A]\!] = \{[\varepsilon \oplus [\![\Delta]\!]][\![A]\!]\}$$

to value types, computation types, pegs, polytypes, interfaces, abilitites, adjustments, and type environments. Ports elaborate to suspended computation types.

The translation on terms elaborates pattern matching, which depends on the type of the term, so we specify it as a translation on derivation trees. Without loss of generality, we only write the judgement at the root of a derivation and only write the term when referring to an descendent of the root. The translation on derivations is the homomorphic extension of the following equations on multihandler definitions

$$
\begin{array}{l}
[\![\Gamma\ [\Sigma]\vdash \{e\} : C]\!] = [\![e]\!] \\
[\![\Gamma \vdash \{((r_{i,j})_j \mapsto n_i)_i\} : (T_j \to)_j\ G]\!] = \\
\qquad [\![\Gamma]\!]\ [[\![\Sigma]\!]\vdash \lambda(x_j)_j.P((x_j)_j, ([\![T_j]\!])_j, ((r_{i,j})_j \mapsto [\![n_i]\!])) \\
\qquad\qquad : \{([\![T_j]\!] \to)_j\ [\![G]\!]\}, \qquad \text{each } x_j \text{ is fresh}
\end{array}
$$

where $P(\overline{x}, \overline{A}, \overline{u})$ is a function that takes a list of variables $\overline{x}$ to eliminate, a list of pattern types $\overline{R}$ (pattern types R are either value types A or port types T), and a pattern matching matrix $\overline{u}$, and yields a Core Frank term.

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 P 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 P. 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.

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 operator $us\ @\ r$ 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 multihandlers are elaborated into Core Frank, we give the Core Frank representation of the `pipe` multihandler.

```
letrec pipe :
  ∀ε X Y.
    ⟨ι + Send X⟩Unit → ⟨ι + Receive X⟩Y → [ε + Abort]Y
      = λx y.handle x with
                ⟨send x → g⟩ ↦
                  handle y with
                    ⟨receive → r⟩ ↦ pipe (s unit) (r x)
                  | y             ↦ y
              | x ↦
                case x of unit ↦
                  handle y with
                    ⟨receive → r⟩ ↦ abort!
                  | y             ↦ y
```

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.

Our pattern matching elaboration procedure is quite direct, but is not at all optimised for efficiency. We believe it should be reasonably straightforward to adapt standard techniques (e.g. [24]) 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.)

Pattern matching elaboration preserves typing.

THEOREM 1.
If $\Gamma\ [\Sigma]\vdash m \Rightarrow A$ *then* $[\![\Gamma]\!]\ [[\![\Sigma]\!]\vdash [\![m]\!] \Rightarrow [\![A]\!]$.
If $\Gamma\ [\Sigma]\vdash n : A$ *then* $[\![\Gamma]\!]\ [[\![\Sigma]\!]\vdash [\![A]\!] \Rightarrow [\![n]\!]$.

4.2 Incomplete and Ambiguous Pattern Matching as Effects

The function P provides a straightforward means for checking coverage and redundancy of pattern matching. Incomplete coverage can occur iff P is invoked on three empty lists $P([\,], [\,], [\,])$, which means P is undefined on its input. Redundancy occurs iff the final clause defining P 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. The former may be permitted if the ambient ability contains the $Abort$ signature, in which case incomplete patterns are translated into the $aborting : Zero$ command, which can then be handled however the programmer wishes. Similarly, we can define a $choice : Bool$ command, in order to allow ambiguous pattern matches to be handled by the programmer. We discuss this possibility further in section 7.

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

$$\frac{x : A \in \Gamma}{\Gamma\,[\Sigma]\vdash x \Rightarrow A} \qquad \frac{f : \forall \overline{Z}.A \in \Gamma \qquad \text{if } \varepsilon \in dom(\theta) \text{ then } \theta(\varepsilon) = \Sigma}{\Gamma\,[\Sigma]\vdash f \Rightarrow \theta(A)} \qquad \frac{c : \overline{A} \to B \in \Sigma}{\Gamma\,[\Sigma]\vdash c \Rightarrow \{\overline{A} \to\}\,[\Sigma]B}$$

$$\frac{\Gamma\,[\Sigma]\vdash m \Rightarrow \{\overline{A \to}\,[\Sigma]B\} \qquad \overline{\Gamma\,[\Sigma]\vdash n : A}}{\Gamma\,[\Sigma]\vdash m\,\overline{n} \Rightarrow B} \qquad \frac{\Gamma\,[\Sigma]\vdash n : A \qquad \text{if } \varepsilon \in dom(\theta) \text{ then } \theta(\varepsilon) = \Sigma}{\Gamma\,[\Sigma]\vdash (n : \forall \overline{Z}.A) \Rightarrow \theta(A)}$$

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

$$\frac{\Gamma\,[\Sigma]\vdash m \Rightarrow A \qquad A = B}{\Gamma\,[\Sigma]\vdash m : B} \qquad \frac{\overline{\Gamma\,[\Sigma]\vdash n : A} \qquad k\,\overline{A} \in D\,\Sigma^?\,\overline{B}}{\Gamma\,[\Sigma]\vdash k\,\overline{n} : D\,\Sigma^?\,\overline{B}} \qquad \frac{\Gamma, \overline{x} : \overline{A}\,[\Sigma']\vdash n : B}{\Gamma\,[\Sigma]\vdash \lambda\overline{x}.n : \{\overline{A \to}\,[\Sigma']B\}}$$

$$\frac{\Gamma\,[\Sigma]\vdash m \Rightarrow D\,\Sigma^?\,\overline{B} \qquad (\Gamma, \overline{x} : \overline{A}\,[\Sigma]\vdash n : B')_{k\,\overline{A} \in D\,\Sigma^?\,\overline{B}}}{\Gamma\,[\Sigma]\vdash \mathbf{case}\ m\ \mathbf{of}\ (k\,\overline{x} \mapsto n)_k : B'}$$

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

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

Figure 4. Core Frank Typing Rules

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

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

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

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

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

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

$$\begin{aligned}
PatternTypes(D\,\Sigma^?\,As, k) &= Bs, && \text{where } \mathcal{D}(D\,\Sigma^?\,As, k) = Bs \\
PatternTypes(I\ As, c) &= B' :: Bs, && \text{where } \mathcal{I}(I\ As, c) = Bs \to B'
\end{aligned} \qquad \begin{aligned}
PatternTypes(\iota) &= [] \\
PatternTypes(\Delta + I\ As, c) &= \\
&PatternTypes(\Delta, c) +\!\!+ PatternTypes(I\ As, c)
\end{aligned}$$

Figure 5. Pattern Matching Elaboration

5. Dynamic 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} [15]. 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 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. We write $n[m/x]$ for n with m substituted for x and similarly $n[\overline{m}/\overline{x}$ for n with the multiple simultaneous each m_i substituted for x_i. 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 $HC(E)$ for the set of commands handled by evaluation context E. The side condition on the command rule ensures that command c' is handled by the nearest enclosing handler that has a clause for handling c'. A more intensional way to achieve the same behaviour would be to explicitly forward unhandled commands using an additional rule.

Reduction preserves typing.

THEOREM 2 (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 (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 3 (Normal Forms). *We say that a checkable term* n *is normal if it is either a value* v *or of the form* $E[c\ \overline{v}]$ *where* $c \notin HC(E)$.

THEOREM 4 (Type Soundness).
If $\cdot\ [\Sigma]\vdash n : A$ *then either* n *is normal 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 datatypes have been entirely first order, but our type system admits datatypes which abstract over an ability 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 [39].

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 (or whatever) was doing when the console actions happened.

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

Note that the empty *brackets* may all be omitted. We can infer the need for an ability parameter just from the presence of the *braces*, which show that `Log X` stores computations whose pegs must refer to an ability. We have included the brackets to emphasise where that ability is used, when caching the initial computation, and when caching the continuation corresponding to each input action.

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`.

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
```

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

$$
\begin{array}{rl}
(\lambda \overline{x}.n : P)\ \overline{v} & \longrightarrow n[\overline{v}/\overline{x}] \\
\mathbf{case}\ k'\ \overline{v}\ \mathbf{of}\ (k\ \overline{x_k} \mapsto n_k)_k & \longrightarrow n_{k'}[\overline{v}/\overline{x_{k'}}] \\
\mathbf{let}\ f : P = v\ \mathbf{in}\ n & \longrightarrow n[(v : P)/f] \\
\mathbf{letrec}\ \overline{f : P = \lambda \overline{x}.n}\ \mathbf{in}\ n' & \longrightarrow n'[(\lambda \overline{x}.\mathbf{letrec}\ \overline{f : P = \lambda \overline{x}.n}\ \mathbf{in}\ n : P)/f] \\
\mathbf{handle}^{\Delta}\ v\ \mathbf{with}\ (c\ \overline{x_c} \to g_c \mapsto n_c)_c \mid x \mapsto n' & \longrightarrow n'[v/x] \\
\mathbf{handle}^{\Delta}\ E[c'\ \overline{v}]\ \mathbf{with}\ (c\ \overline{x_c} \to g_c \mapsto n_c)_c \mid x \mapsto n' & \longrightarrow n_{c'}[\overline{v}/\overline{x_{c'}}, \lambda z.E[z]/g_{c'}], \quad \text{if } c' \notin HC(E)
\end{array}
$$

$$
\frac{m \longrightarrow m'}{E[m] \longrightarrow E[m']} \qquad \frac{n \longrightarrow n'}{E[n] \longrightarrow E[n']}
$$

Figure 6. Small-Step Operational Semantics for Core Frank

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 ability polymorphism—a topic of active investigation.

7. Related Work

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

Effect Handler Implementations A natural implementation for handlers is to use *free monads* [15]. Swierstra [41] illustrates how to write effectful programs with free monads in Haskell, taking advantage of type-classes to provide a certain amount of modularity.

Wu and Schrijvers [47] show how to obtain a particularly efficient implementation of deep handlers taking advantage of fusion. Their work explains how Kammar et al. [15] obtain strong performance.

Kiselyov and Ishii [16] optimise their shallow effect handlers implementation, which is based on free monads, by taking advantage of an efficient representation of sequences of monadic operations [44].

The experimental multicore extension to OCaml [7] 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.

Languages other than Frank that attempt to elide some effect variables from source code include Links [21] and Koka [18]. Neither eliminates effect variables altogether. Recently, Hillerström and Lindley [11, 12] have implemented an extension of Links to support effect handlers.

Layered Monads and Monadic Reflection Inspired by Bauer and Pretnar's Eff, Visscher has implemented the `effects` library [45]. The key idea is to layer continuation monads in the style of Filinski [8], using Haskell type classes to automatically infer lifting between layers.

Filinski's work on monadic reflection [9] and layered monads [8] 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 [40] 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.

Variations and Applications Lindley [20] investigates an adaptation of effect handlers to more restrictive forms of computation based on idioms [27] and arrows [14]. Wu et al. [48] 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 [16] provide solutions to particular instances of this problem. Schrijvers et al. [38] apply effect handlers to logic programming.

8. Future Work

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

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.

Failure and Choice in Pattern Matching Patterns do not always match, and if we were to allow operators such as `append` (or, more generally, the plugging together of n-hole contexts), they can match in multiple ways, delivering a searchable solution space. Operationally, pattern matching can be seen as the means to compute a value environment (interpreting an interface of pattern variables), whilst potentially aborting or making choices to navigate search. It seems feasible to mediate failure and choice effects as effects, separating what it is to *be* a solution from the strategy used to *find* one. Wu, Schrijvers and Hinze [48] 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 [5], 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 [4]. We would like to explore adding resources or a similar abstraction to Frank.

Explicit Effect Polymorphism In this paper, we have shown the high power-to-weight ratio of working relative to an ambient ability which operators silently abstract. However, we expect to reach the expressive limits of our system soon. E.g., while we can easily define composition of *functional* operators, we cannot yet give a *general* type for the composition of operators which make arbitrary adjustments to the ambient ability. We do not imagine that a simple, predictable type discipline can account for such definitions with the same degree of silence.

Our focus has been on making common use cases convenient, but in the longer term, we should support programmers who are willing to be explicit about more sophisticated management of effectful abilities. Internally, we do manifest the ambient ability as an abstracted variable, so we have a basis for more general abstraction over abilities. The design question then becomes how best to deliver that power in the surface language and manage the constraints which thus arise with the minimum of fuss. We similarly need to account for negative adjustments (removing specific interfaces) and nugatory adjustments (reverting to purity). We should also consider allowing handlers to trap some or even all commands generically, just as long as their ports make this possibility clear. Secret interception of signals 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 [25] observes that indexing input and output types over the state effectively lets us specify interfaces in a proof-relevant Hoare logic. Hancock and Hyvernat [10] 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 usage 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 [23] and Haskell type classes [46].

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 [2].

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 [26].

Implementation Our progress on direct-style programming with locally managed effects has created a rapidly moving target for our implementation efforts, but we must catch up and deliver a prototype which fits with the account in this paper and acts not only as a springboard for further study, but also as tool for tackling the programming problems we face in real life. Whether we are writing elaborators for advanced programming languages, or 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.

References

[1] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In Morrisett and Uustalu [28], pages 185–196. URL `http://doi.acm.org/10.1145/2500365.2500591`.

[2] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In Morrisett and Uustalu [28], pages 197–208. URL `http://doi.acm.org/10.1145/2500365.2500597`.

[3] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014. URL `http://dx.doi.org/10.2168/LMCS-10(4:9)2014`.

[4] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015. URL `http://dx.doi.org/10.1016/j.jlamp.2014.02.001`.

[5] E. Brady. Programming and reasoning with algebraic effects and dependent types. In Morrisett and Uustalu [28], pages 133–144. URL `http://doi.acm.org/10.1145/2500365.2500581`.

[6] M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors. *Proceedings of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, 2011. ACM.

[7] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects, 9 2015. OCaml Workshop.

[8] A. Filinski. Representing layered monads. In A. W. Appel and A. Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 175–188. ACM, 1999. URL http://doi.acm.org/10.1145/292540.292557.

[9] A. Filinski. Monads in action. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 483–494. ACM, 2010. URL http://doi.acm.org/10.1145/1706299.1706354.

[10] P. Hancock and P. Hyvernat. Programming interfaces and basic topology. *Ann. Pure Appl. Logic*, 137(1-3):189–239, 2006. URL http://dx.doi.org/10.1016/j.apal.2005.05.022.

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

[12] D. Hillerström and S. Lindley. Liberating effects with rows and handlers, June 2016. Draft http://homepages.inf.ed.ac.uk/slindley/papers/links-effect-draft-june2016.pdf.

[13] R. Hinze and J. Voigtländer, editors. *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, 2015. Springer. URL http://dx.doi.org/10.1007/978-3-319-19797-5.

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

[15] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In Morrisett and Uustalu [28], pages 145–158. URL http://doi.acm.org/10.1145/2500365.2500590.

[16] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM, 2015. URL http://doi.acm.org/10.1145/2804302.2804319.

[17] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In C. Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013. URL http://doi.acm.org/10.1145/2503778.2503791.

[18] D. Leijen. Koka: Programming with row polymorphic effect types. In P. Levy and N. Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.*, volume 153 of *EPTCS*, pages 100–126, 2014. URL http://dx.doi.org/10.4204/EPTCS.153.8.

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

[20] S. Lindley. Algebraic effects and effect handlers for idioms and arrows. In J. P. Magalhães and T. Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 47–58. ACM, 2014. URL http://doi.acm.org/10.1145/2633628.2633636.

[21] S. Lindley and J. Cheney. Row-based effect types for database integration. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 91–102. ACM, 2012. URL http://doi.acm.org/10.1145/2103786.2103798.

[22] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 47–57. ACM Press, 1988. URL http://doi.acm.org/10.1145/73560.73564.

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

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

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

[26] C. McBride. Turing-completeness totally free. In Hinze and Voigtländer [13], pages 257–275. URL http://dx.doi.org/10.1007/978-3-319-19797-5_13.

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

[28] G. Morrisett and T. Uustalu, editors. *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, 2013. ACM. URL http://dl.acm.org/citation.cfm?id=2500365.

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

[30] G. D. Plotkin and J. Power. Semantics for algebraic operations. *Electr. Notes Theor. Comput. Sci.*, 45:332–345, 2001. URL http://dx.doi.org/10.1016/S1571-0661(04)80970-8.

[31] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. URL http://dx.doi.org/10.1007/3-540-45315-6_1.

[32] G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. URL http://dx.doi.org/10.1007/3-540-45931-6_24.

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

[34] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. URL http://dx.doi.org/10.1007/978-3-642-00590-9_7.

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

[36] M. Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014. URL http://dx.doi.org/10.2168/LMCS-10(3:21)2014.

[37] M. Sato, T. Sakurai, and R. M. Burstall. Explicit environments. *Fundam. Inform.*, 45(1-2):79–115, 2001. URL http://content.iospress.com/articles/fundamenta-informaticae/fi45-1-2-05.

[38] 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, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 259–270. ACM, 2014. URL http://doi.acm.org/10.1145/2643135.2643145.

[39] O. Shivers and A. J. Turon. Modular rollback through control logging: a pair of twin functional pearls. In Chakravarty et al. [6], pages 58–68. URL http://doi.acm.org/10.1145/2034773.2034783.

[40] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In Chakravarty et al. [6], pages 15–27. URL http://doi.acm.org/10.1145/2034773.2034778.

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

[42] W. Swierstra, editor. *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, 2014. ACM. URL http://dl.acm.org/citation.cfm?id=2633357.

[43] J. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994. . URL http://dx.doi.org/10.1006/inco.1994.1046.

[44] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In Swierstra [42], pages 133–144. URL http://doi.acm.org/10.1145/2633357.2633360.

[45] S. Visscher. The effects package (0.2.2), 2012. http://hackage.haskell.org/package/effects.

[46] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. URL http://doi.acm.org/10.1145/75277.75283.

[47] N. Wu and T. Schrijvers. Fusion for free - efficient algebraic effect handlers. In Hinze and Voigtländer [13], pages 302–322. URL http://dx.doi.org/10.1007/978-3-319-19797-5_15.

[48] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In Swierstra [42], pages 1–12. URL http://doi.acm.org/10.1145/2633357.2633358.