# Do Be Do Be Do

Sam Lindley
The University of Edinburgh

Conor McBride
University of Strathclyde

## Abstract

We explore the design of Frank, a strict functional programming language designed from the ground up around a novel variant of Plotkin and Pretnar's *effect handler* abstraction. Inspired by Levy's call-by-push-value calculus, Frank makes an explicit distinction between computation and value types. Frank combines the advantages of call-by-push-value, call-by-value, and effect typing.

## 1. Introduction

Plotkin and Power's algebraic effects [19–22], in conjunction with Plotkin and Pretnar's handlers for algebraic effects [23, 24], provide a compelling foundation for effectful programming. By separating effect signatures from their implementation, algebraic effects provide a high degree of modularity, allowing programmers to express effectful programs independently of the concrete interpretation of their effects. A handler is an interpretation of the effects of an algebraic computation.

Effect handlers in Frank generalise functions. An effect handler acts as an interpreter for a specified set of commands whose signatures are statically tracked by the type system. A function is the special case of an effect handler whose 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* effects.
- The combination of pattern matching and effect handlers in such a way that incomplete or ambiguous pattern matching can be realised as concrete effects that can be handled however the programmer chooses using further effect handlers.
- Multi-handlers as both an abstraction for handling multiple computations over different effect sets simultaneously and a characterisation of effect-handlers as generalised functions.
- A description of pattern matching compilation from Frank into a core language, *Core Frank*.
- A translation from Core Frank into $F_{\text{eff}}$, a polymorphic variant of the $\lambda_{\text{eff}}$ calculus of Kammar et al [8], which in turn extends Levy's call-by-push-value calculus [11].
- A straightforward small-step operational semantics for $F_{\text{eff}}$, yielding, in combination with the translation to $F_{\text{eff}}$, a type soundness result for Frank.



A number of other languages and libraries are built around effect handlers and algebraic effects.

Bauer and Pretnar's Eff [2]. A significant difference between Frank and the original version of Eff [2] 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 [25] 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.

Handlers in action [8]. In previous work with Kammar and Oury [8], the first author designed and experimented with 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 et al [9] have designed another Haskell library for effect handlers, making a different collection of design choices.

Brady [4] 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.

In Plotkin and Powers setting, one defines algebraic effects with respect to an equational theory. In all of the above implementations, and in Frank, the equational theory is taken to be the free theory, in which there are no equations.

The second author has been plotting Frank since 2007 [14]. He has implemented a prototype of a previous version of Frank [16]. The design described in the current paper has much in common with that implementation, but there are some syntactic and semantic differences. The most important change in the current design is the introduction of multi-handlers as a generalisation of both functions and handlers.

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 multi-handlers 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 call-by-value embedding of Core Frank into $F_{\text{eff}}$, a variant of Kammar et al's $\lambda_{\text{eff}}$ calculus with shallow handlers, explicit polymorphism and general recursion. Section 6 gives a se-

mantics for $F_{\text{eff}}$, which when composed with pattern matching and the call-by-value embedding, yields a semantics for Frank. Section 7 outlines related work and Section 8 discusses future work.

## 2. Introducing Frank

Frank is a functional programming language with effects and handlers in the style of Eff controlled by a type system inspired by Levy's call-by-push-value [11]. Doing and Being are clearly separated, and managed by distinguished notions of computation and value types.

Concrete values live in inductive datatypes.

```
data List X
   = nil
   | cons X (List X)

data Zero =

data Unit = unit
```

We can write perfectly ordinary functional programs, with (compulsory) type signatures.

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

Higher-order functions are passed suspended computations. Braces are "suspenders".

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

A value type `V` is a datatype `D V1 ... Vn`, a suspended computation type `{C}`, or a type variable `X`. A computation type can be a function type `V -> C` or an effect-annotated value type `[S1 ... Sn]V`, where an empty bracket may be omitted.

Effects are collections of signatures, which describe a choice of commands. Here are some simple signatures.

```
sig Send X
  = send : Unit(X)

sig Receive X
  = receive : X

sig 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 abort command returns an element of the empty type, `Zero`.

Frank is a call-by-value language, but it naturally distinguishes `[]V`, the type of suspended pure computations which deliver a `V`, from `V` itself. We can thus define a kind of "semicolon" just as the function which ignores its first argument.

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

Frank has effect polymorphism, enough to allow higher-order functions to pass effect permissions to their parameters. The following uses map to send a list of things, one at a time.

```
sends : List X -> [Send X]Unit
sends xs = semi (map send xs) unit
```

The reason this type checks at all is because `map` is implicitly polymorphic in its effects.

The following does not typecheck, because the `Send` effect is not permitted in the return type of `bad`.

```
bad : List X -> Unit
bad xs = semi (map send xs) unit
```

Writing control operators is not too tricky.

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

We can use `bind` to define a polymorphic `abort` function.

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

The term `{}` denotes a suspended computation containing an empty collection of pattern matching clauses covering the `Zero` return type of `aborting`.

Here is a computation which receives and concatenates lists until one is empty.

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

The command `receive` is a suspended computation of type `{[Receive (List X)]List X}` that delivers a list when forced. The notation `receive!` forces a thunk.

If `f` is a suspended computation of function type, then `f x` is syntactic sugar for `f! x`. Thus we need only explicitly force computations that take no arguments.

Effects are handled by special functions called *effect handlers*. Effect handlers in Frank can take multiple computations as arguments, hence they are *multi-handlers*. In fact standard functions like the ones we have seen so far are just special cases of multi-handlers in which the all the arguments are pure.

A multi-handler has type `R1 -> ... -> Rm -> R` where each `Ri` and `R` is an effect-annotated value type. For instance, we can write a `pipe` multi-handler which handles `send` commands from one computation by matching them against corresponding `receive` commands from another.

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

The type signature conveys several different things. The `pipe` handler must handle all `Send X` commands in its first argument and all `Receive X` commands in its second argument. The first argument returns values of type `Unit` and the second argument returns values of type `Y`. The handler itself is allowed to perform `Abort` commands and returns a final value of type `Y`.

Here are some things to send.

```
hello : List Char
hello =
  cons 'h' (cons 'e' (cons 'l'
    (cons 'l' (cons 'o' nil))))

space : List Char
space = cons ' ' nil

world : List Char
world =
  cons 'w' (cons 'o' (cons 'r'
    (cons 'l' (cons 'd' nil))))
```

Types

$$
\begin{array}{ll}
\text{(values)} & U, V ::= D\ \overline{U} \mid \{C\} \mid X \\
\text{(computations)} & C ::= R \mid R \to C \\
\text{(returners)} & R ::= [\Sigma]V \\
\text{(quantifiers)} & Z ::= X \mid \varepsilon \\
\text{(polytypes)} & P ::= \forall \overline{Z}.Q \\
\text{(thunks)} & Q ::= \{C\} \\
\text{(signatures)} & S\ \overline{X} ::= \cdot \mid c : U(\overline{V}), S\ \overline{X} \\
\text{(effects)} & \Sigma ::= \varnothing \mid \Sigma, S\ \overline{V} \mid \varepsilon \\
\text{(type environments)} & \Gamma ::= \cdot \mid \Gamma, x : V \mid f : P
\end{array}
$$

Terms

$$
\begin{array}{ll}
\text{(inferable values)} & u ::= x \mid f \mid c \mid d \\
\text{(checkable values)} & v ::= u \mid k\ \overline{v} \mid \{e\} \\
\text{(inferable computations)} & d ::= u! \mid d\ e \\
 & \quad \mid \mathsf{letrec}\ \overline{f : P = e}\ \mathsf{in}\ d \\
\text{(checkable computations)} & e ::= v \mid \overline{r} \mapsto e \mid () \mid e \vdash e \\
\text{(value patterns)} & p ::= x \mid k\ \overline{p} \\
\text{(computation patterns)} & r ::= p \mid c\ \overline{p}\ ?\ g \mid x!
\end{array}
$$

**Figure 1.** Frank Syntax

Here is a computation which sends them.

```
sender : [Send (List Char)]Unit
sender =
  sends (cons (hello
           (cons space (cons world nil))))
```

Here is a `main` function, which plugs `sender` and `catter` together and sends their output to the console.

```
main : [Abort, Console](List Unit)
main = map ouch (pipe sender catter)
```

where the `Console` operations are handled specially at the top-level according to the following signature.

```
sig Console
  = inch : Char(Unit)
  | ouch : Unit(Char)
```

The type system does two separate things:

- It ensures that value types coincide.
- It ensures that effects required are included in effects enabled.

The fun of Frank is that one can say what it is to *be* a computation without saying what it is to *do* it. Doing and being are separately negotiable, and readily interleaved in different ways. Or as Frank Sinatra put it,

do be do be do

## 3. Type System

In this section we give a formal presentation of Frank's type system. The syntax of Frank types and terms is given in Figure 1. The types are divided into value types and computation types in a similar fashion to Levy's call-by-push-value calculus [11]. Value types are datatypes ($D\ \overline{U}$), suspended $C$ computations ($\{C\}$), otherwise known as *thunks*, or type variables ($X$).

Computation types are constructed from *returners*. A returner ($[\Sigma]V$) represents the type of a computation that returns values of type $V$ while performing effects in $\Sigma$. In general, computation types represent multi-handlers. The type $[\Sigma_1]V_1 \to \cdots \to [\Sigma_n]V_n \to [\Sigma]V$, is the type of an $n$-handler. For each argument type $[\Sigma_i]V_i$ the multi-handler must handle effects in $\Sigma_i$ on that argument. Such an $n$-handler handles all of its arguments simultaneously. As a result of handling its arguments it returns a value of type $V$ and may perform effects in $\Sigma$. We often write $\overline{[\Sigma]V} \to C$ as an abbreviation for $[\Sigma_1]V_1 \to \cdots \to [\Sigma_n]V_n \to C$.

Polytypes are restricted to thunks. Effect polymorphism is restricted to a single effect variable $\varepsilon$, which in practice Frank programmers need never write.

Datatypes and effect signatures are declared at the top-level. An effect signature $S\ \overline{X}$ consists of a collection of command declarations of the form $c : U(\overline{V})$, denoting that command $c$ takes arguments of types $\overline{V}$ and returns a value of type $U$. The types $\overline{V}$ and $U$ may all depend on $\overline{X}$. Each command many appear only once in a signature, and each command may appear in only one signature.

An effect set is a sequence of signatures initiated either with the empty effect $\varnothing$ (yielding a *closed effect set*) or the only effect variable $\varepsilon$ (yielding an *open effect set*). Order is important, as repeats are permitted, in which case the right-most signature overrides all others with the same name.

Shadowing arises naturally in two ways. First, given an open effect set $\Sigma$ we may substitute an arbitrary effect set for $\varepsilon$. Second, we define a notion of effect extension: $\Sigma \oplus \Sigma'$ is the *extension* of effect set $\Sigma$ with closed effect set $\Sigma'$, formally:

$$
\begin{array}{lll}
\Sigma & \oplus & \varnothing = \Sigma \\
\Sigma & \oplus & (\Sigma', S\ \overline{V}) = (\Sigma \oplus \Sigma'), S\ \overline{V}
\end{array}
$$

Type environments distinguish monomorphic and polymorphic variables.

Just as with the types, Frank terms are separated into value terms and computation terms. With future extensions, such as dependent types, in mind, Frank adopts a bidirectional typing discipline [18]. Thus terms are further sub-divided into those whose type is inferable, and those that may be checked against a type.

Frank is less strict about the separation between value and computation terms than call-by-push-value is. For instance, inferable computations can sometimes be treated as inferable values. This is a deliberate design decision, with the aim of making Frank convenient to program with.

The typing rules for Frank are given in Figure 2. The judgement $\Gamma\ [\Sigma] \vdash u\ \mathbf{infers}\ V$ says that given type environment $\Gamma$, ambient effects $\Sigma$, and inferable value term $u$, then we can *infer* that the type of $u$ is $V$. The judgement $\Gamma\ [\Sigma] \vdash V\ \mathbf{checks}\ v$ says that given type environment $\Gamma$, ambient effects $\Sigma$, value type $V$, and checkable value term $v$, then we can *check* that $V$ is the type of $v$. The judgement $\Gamma\ [\Sigma] \vdash d\ \mathbf{infers}\ C$ says that given type environment $\Gamma$, ambient effects $\Sigma$, and inferable computation term $d$, then we can *infer* that the type of $d$ is $C$. The judgement $\Gamma \vdash C\ \mathbf{checks}\ e$ says that given type environment $\Gamma$, computation type $C$, and checkable computation term $e$, then we can *check* that $C$ is the type of $e$.

The types of monomorphic variables ($x$) are simply looked up in the type environment. The types of polymorphic variables ($f$) are looked up and instantiated. In practice this means applying a simple unification-based algorithm. The type of a command ($c$) is looked up from the ambient effects. An inferable computation ($d$) is also an inferable value, providing it is a returner whose effects agree with the ambient effects.

Any inferable value ($u$) is also checkable against its inferred type. Datatype ($k\ \overline{v}$) and thunk ($\{e\}$) terms are checkable by checking their components. The side condition on the thunk introduction rule requires that the pattern matching clauses of $e : C$ cover $C$.

A thunk $u$ can by forced ($u!$) if its inferred type agrees with the ambient effects. To infer the type of a handler application $d\ e$, we first infer the type of $d$, and then check that the argument matches

$\boxed{\Gamma\,[\Sigma]\vdash u\ \mathbf{infers}\ V}$

$$\frac{x : V \in \Gamma}{\Gamma\,[\Sigma]\vdash x\ \mathbf{infers}\ V} \qquad \frac{f : \forall \overline{Z}.Q \in \Gamma \qquad dom(\theta) = \{\overline{Z}\} \qquad \varepsilon \in dom(\theta) \implies \theta(\varepsilon) = \Sigma}{\Gamma\,[\Sigma]\vdash f\ \mathbf{infers}\ \theta(Q)} \qquad \frac{c : U(\overline{V}) \in \Sigma}{\Gamma\,[\Sigma]\vdash c\ \mathbf{infers}\ \{\overline{[]V} \to [\Sigma]U\}}$$

$$\frac{\Gamma\,[\Sigma]\vdash d\ \mathbf{infers}\ [\Sigma]V}{\Gamma\,[\Sigma]\vdash d\ \mathbf{infers}\ V}$$

$\boxed{\Gamma\,[\Sigma]\vdash V\ \mathbf{checks}\ v}$

$$\frac{\Gamma\,[\Sigma]\vdash u\ \mathbf{infers}\ U \qquad U = V}{\Gamma\,[\Sigma]\vdash V\ \mathbf{checks}\ u} \qquad \frac{(\Gamma\,[\Sigma]\vdash V_i\ \mathbf{checks}\ v_i)_i \qquad k\ \overline{V} \in D\ \overline{U}}{\Gamma\,[\Sigma]\vdash D\ \overline{U}\ \mathbf{checks}\ k\ \overline{v}} \qquad \frac{\Gamma \vdash C\ \mathbf{checks}\ e \qquad e\ \text{covers}\ C}{\Gamma\,[\Sigma]\vdash \{C\}\ \mathbf{checks}\ \{e\}}$$

$\boxed{\Gamma\,[\Sigma]\vdash d\ \mathbf{infers}\ C}$

$$\frac{\Gamma\,[\Sigma]\vdash u\ \mathbf{infers}\ \{C\} \qquad C\ \mathbf{does}\ \Sigma}{\Gamma\,[\Sigma]\vdash u!\ \mathbf{infers}\ C} \qquad \frac{\Gamma\,[\Sigma]\vdash d\ \mathbf{infers}\ [\Sigma']V \to C \qquad \Gamma \vdash [\Sigma \oplus \Sigma']V\ \mathbf{checks}\ e}{\Gamma\,[\Sigma]\vdash d\ e\ \mathbf{infers}\ C}$$

$$\frac{(\Gamma, \overline{f : \forall \overline{Z}.Q} \vdash e_i\ \mathbf{checks}\ Q_i)_i \qquad \Gamma, \overline{f : \forall \overline{Z}.Q}\,[\Sigma]\vdash d\ \mathbf{infers}\ C}{\Gamma\,[\Sigma]\vdash \mathsf{letrec}\ \overline{f : \forall \overline{Z}.Q = e}\ \mathsf{in}\ d\ \mathbf{infers}\ C}$$

$\boxed{\Gamma \vdash C\ \mathbf{checks}\ e}$

$$\frac{\Gamma\,[\Sigma]\vdash V\ \mathbf{checks}\ v}{\Gamma \vdash [\Sigma]V\ \mathbf{checks}\ v} \qquad \frac{\overline{\Sigma'}\ \text{closed} \qquad ([\Sigma'_i]V_i\ \mathbf{matches}\ r_i \dashv \Delta_i)_i \qquad \Gamma, \overline{\Delta} \vdash C\ \mathbf{checks}\ e}{\Gamma \vdash \overline{[\Sigma']V} \to C\ \mathbf{checks}\ \overline{r} \mapsto e} \qquad \frac{V\ \text{uninhabited}}{\Gamma \vdash [\Sigma']V \to C\ \mathbf{checks}\ ()}$$

$$\frac{\Gamma \vdash R \to C\ \mathbf{checks}\ e \qquad \Gamma \vdash R \to C\ \mathbf{checks}\ e'}{\Gamma \vdash R \to C\ \mathbf{checks}\ e \shortmid e'}$$

$\boxed{C\ \mathbf{does}\ \Sigma}$

$$\frac{}{\overline{[\Sigma]V}\ \mathbf{does}\ \Sigma} \qquad \frac{C\ \mathbf{does}\ \Sigma}{R \to C\ \mathbf{does}\ \Sigma}$$

$\boxed{V\ \mathbf{matches}\ p \dashv \Delta}$

$$\frac{}{V\ \mathbf{matches}\ x \dashv x : V} \qquad \frac{k\ \overline{V} \in D\ \overline{U} \qquad (V_i\ \mathbf{matches}\ p_i \dashv \Delta_i)_i}{D\ \overline{U}\ \mathbf{matches}\ k\ \overline{p} \dashv \overline{\Delta}}$$

$\boxed{R\ \mathbf{matches}\ r \dashv \Delta}$

$$\frac{V\ \mathbf{matches}\ p \dashv \Delta}{[\Sigma]V\ \mathbf{matches}\ p \dashv \Delta} \qquad \frac{c : U(\overline{V}) \in \Sigma \qquad (V_i\ \mathbf{matches}\ p_i \dashv \Delta_i)_i}{[\Sigma]V\ \mathbf{matches}\ c\ \overline{p}\ ?\ g \dashv \overline{\Delta}, g : \{[\varnothing]U \to [\Sigma]V\}} \qquad \frac{}{[\Sigma]V\ \mathbf{matches}\ x! \dashv x : \{[\Sigma]V\}}$$

**Figure 2.** Frank Typing Rules

the inferred argument type. Note that the effect set on an argument must be closed, and thus we may extend the ambient effect set with the argument effect set when checking the argument. The term $\mathsf{letrec}\ \overline{f : P = e}\ \mathsf{in}\ d$ binds the mutually recursive polymorphic functions $\overline{f : P}$ in $d$.

Any term $v$ that type checks with ambient effects $\Sigma$ at value type $V$ also type checks at computation type $[\Sigma]V$. A multi-handler of type $\overline{[\Sigma']V} \to C$ is built by composing clauses of the form $\overline{r} \mapsto e$, where $\overline{r}$ is a sequence of *computation patterns* whose variables are bound in $e$. The $e \shortmid e'$ construct composes the clauses of $e$ with those of $e'$. A multi-handler is defined by a collection of clauses that provides complete coverage of the input types. The $()$ construct allows empty sets of clauses to be constructed in the event that the value type of a function argument is uninhabited.

We preclude composition of non-clauses (i.e. returners) by constraining $e \shortmid e'$ to have function type. It is perfectly legitimate to compose clauses with different numbers of arguments, though as a preliminary part of pattern matching compilation, we first transform the program to ensure that all composed clauses do have the same number of arguments.

Value patterns are standard, consisting of variable patterns ($x$) and datatype constructor patterns ($k\ \overline{p}$).

Computation patterns are more interesting. Any value pattern $p$ is also a computation pattern (used to match against the value returned by the returner computation). A request pattern $c\ \overline{p}\ ?\ g$ matches against a computation of the form $c!\ \overline{v}$ if the values $v$ match against $p$. Furthermore, it also binds $g$ to the continuation of the computation delimited by the nearest enclosing multi-handler. This is where the real power of multi-handlers arises. A thunk pattern $x!$ matches any computation reifying it as a thunk bound to $x$.

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

$$\begin{array}{l}\mathsf{let}\ x = e\ \mathsf{in}\ e' \equiv \\ \quad \mathsf{letrec}\ (bind : \forall \varepsilon\ X\ Y.X \to \{X \to [\varepsilon]Y\} \to [\varepsilon]Y) = \\ \quad\quad x\ f\ \mapsto\ f\ x\ \mathsf{in}\ bind!\ e\ \{x \mapsto e\}\end{array}$$

### 3.1 Effect Polymorphism with an Invisible Effect Variable

We now give a brief description of how syntactic sugar allows Frank programmers to omit effect variables completely. Consider the type of `map` in Section 2:

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

Apart from perhaps the curly braces, this looks pretty much the same as the 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:

$$\{[\varnothing]X \to [\varepsilon]Y\} \to [\varnothing](List\ X) \to [\varepsilon](List\ Y)$$

Let us distinguish between returners $[\Sigma]V$ in argument position, whose effects $\Sigma$ must be closed, and those in tail position, whose effects need not be closed. We call the former *ports*, and the latter *pegs*.

Observe that ports are closed, so it is never necessary to write the $\varnothing$ in their effects. Pegs, on the other hand, may be open or closed. We adopt the convention that $\varepsilon$ may be omitted from the start of a peg's effects. Thus, if we know it is a peg, then $V$ means $[\,]V$, which means $[\varepsilon]V$, and $[Abort]V$ means $[\varepsilon, Abort]V$.

We now summarise the syntactic sugar. For ports and pegs:

$$V \equiv [\,]V$$

For ports:

$$[\overline{S\ \overline{V}}]U \equiv [\varnothing, \overline{S\ \overline{V}}]U$$

For pegs:

$$[\overline{S\ \overline{V}}]U \equiv [\varepsilon, \overline{S\ \overline{V}}]U$$

With this syntactic sugar in place, we can now avoid writing the effect variable $\varepsilon$ in Frank programs, ever. In addition, we need never write $\varnothing$ in port effect sets. It is sometimes necessary to explicitly write $\varnothing$ in peg effect sets. In particular, a pure top-level program returning values of type $V$ has type $[\varnothing]V$.

## 4. Pattern Matching Compilation

We take a fairly standard approach to compiling away pattern matching. As we may match simultaneously against multiple side-effecting computations, we must be somewhat careful about order. Optionally, we can expose incomplete or ambiguous pattern matching as concrete effects.

The target language of pattern matching compilation, Core Frank, replaces multi-handlers with a combination of call-by-value functions, case statements, and unary effect handlers.

The syntax of Core Frank is given in Figure 3. The Core Frank typing rules are given in Figure 4.

Multi-handlers in Frank become curried functions over suspended computations in Core Frank. Shallow pattern matching on a single request becomes unary effect handling. Shallow pattern matching on a datatype value becomes case analysis. Nested pattern matching on multiple computations is realised as a pattern matching tree constructed from handlers and case statements.

We may adapt standard algorithms for pattern matching compilation apply (e.g. [1] or [13]). Rather than comitting to a particular one, we outline how a pattern matching compiler fits into our setting, what input it takes, and what kind of output it must produce.

Given a Frank expression $\{e\}$ such that $\Gamma \vdash R_1 \ldots R_n \to R\ \mathbf{checks}\ e$ we compile it to an equivalent Core Frank expression $\{(\!|e|\!)\}$. First we expand all of the clauses in $e$ to yield an $n$ column pattern matrix. For instance, suppose the arguments have types $[Send\ Char, Abort]Unit$ and $[Receive\ Char]$, and we have the following clauses:

$$\begin{array}{ll}(send\ x\ ?\ s) & (receive\ ?\ r) \mapsto e_1 \\ (send\ x\ ?\ s) & z \mapsto e_2 \\ (abort\ ?\ s) & e_3 \\ unit & e_4\end{array}$$

then this becomes:

$$\begin{array}{lll}(send\ x\ ?\ s) & (receive\ ?\ r) & \mapsto\ e_1 \\ (send\ x\ ?\ s) & z & \mapsto\ e_2 \\ (abort\ ?\ s) & y! & \mapsto\ bind\ y!\ e_3 \\ unit & y! & \mapsto\ bind\ y!\ e_4\end{array}$$

where $bind$ is defined in Section 2. The missing patterns have been inserted as thunk patterns, which match any computation. Invoking $bind$ allows us to forward the computation bound by the thunk pattern to the existing continuation. Next we generate a vector of fresh variables, one for each argument.

$$x_0\ x_1$$

The goal of pattern matching compilation is to generate a pattern matching tree that matches the variable vector against all of the patterns in the pattern matrix in the correct order.

Types

$$\begin{array}{lll}\text{(values)} & U, V & ::= D\ \overline{U} \mid \{C\} \mid X \\ \text{(computations)} & C & ::= R \mid V \to C \\ \text{(returners)} & R & ::= [\Sigma]V \\ \text{(quantifiers)} & Z & ::= X \mid \varepsilon \\ \text{(polytypes)} & P & ::= \forall \overline{Z}.Q \\ \text{(thunks)} & Q & ::= \{C\} \\ \text{(signatures)} & S\ \overline{X} & ::= \cdot \mid c : U(\overline{V}), S\ \overline{X} \\ \text{(effects)} & \Sigma & ::= \varnothing \mid \Sigma, S\ \overline{V} \mid \varepsilon \\ \text{(type environments)} & \Gamma & ::= \cdot \mid \Gamma, x : V \mid f : P\end{array}$$

Terms

$$\begin{array}{lll}\text{(inferable values)} & u & ::= x \mid f \mid c \mid d \\ \text{(checkable values)} & v & ::= u \mid k\ \overline{v} \mid \{e\} \\ \text{(inferable computations)} & d & ::= u! \mid d\ v \\ & & \mid\ \mathsf{letrec}\ \overline{f : P = e}\ \mathsf{in}\ d \\ \text{(checkable computations)} & e & ::= v \mid \lambda x.e \\ & & \mid\ \mathsf{case}\ u\ \mathsf{of}\ (k\ \overline{x_k} \mapsto e_k)_k \\ & & \mid\ \mathsf{handle}\ d\ \mathsf{with}\ (c\ \overline{x_c}\ ?\ g_c \mapsto e_c)_c \\ & & \quad\ \mid\ x \mapsto e\end{array}$$

**Figure 3.** Core Frank Syntax

$\boxed{C \textbf{ does } \Sigma}$

$$\frac{}{[\Sigma]V \textbf{ does } \Sigma} \qquad \frac{C \textbf{ does } \Sigma}{V \to C \textbf{ does } \Sigma}$$

$\boxed{\Gamma\,[\Sigma]\vdash u \textbf{ infers } V}$

$$\frac{x : V \in \Gamma}{\Gamma\,[\Sigma]\vdash x \textbf{ infers } V} \qquad \frac{f : \forall \overline{Z}.Q \in \Gamma \qquad dom(\theta) = \{\overline{Z}\} \qquad \varepsilon \in dom(\theta) \implies \theta(\varepsilon) = \Sigma}{\Gamma\,[\Sigma]\vdash f \textbf{ infers } \theta(Q)} \qquad \frac{c : U(\overline{V}) \in \Sigma}{\Gamma\,[\Sigma]\vdash c \textbf{ infers } \{\overline{V} \to [\Sigma]U\}}$$

$$\frac{\Gamma\,[\Sigma]\vdash d \textbf{ infers } [\Sigma]V}{\Gamma\,[\Sigma]\vdash d \textbf{ infers } V}$$

$\boxed{\Gamma\,[\Sigma]\vdash V \textbf{ checks } v}$

$$\frac{\Gamma\,[\Sigma]\vdash u \textbf{ infers } U \qquad U = V}{\Gamma\,[\Sigma]\vdash V \textbf{ checks } u} \qquad \frac{(\Gamma\,[\Sigma]\vdash V_i \textbf{ checks } v_i)_i \qquad k\ \overline{V} \in D\ \overline{U}}{\Gamma\,[\Sigma]\vdash D\ \overline{U} \textbf{ checks } k\ \overline{v}} \qquad \frac{\Gamma \vdash C \textbf{ checks } e}{\Gamma\,[\Sigma]\vdash \{C\} \textbf{ checks } \{e\}}$$

$\boxed{\Gamma\,[\Sigma]\vdash d \textbf{ infers } C}$

$$\frac{\Gamma\,[\Sigma]\vdash u \textbf{ infers } \{C\} \qquad C \textbf{ does } \Sigma}{\Gamma\,[\Sigma]\vdash u! \textbf{ infers } C} \qquad \frac{\Gamma\,[\Sigma]\vdash d \textbf{ infers } [\Sigma']V \to C \qquad \Gamma\,[\Sigma \oplus \Sigma']\vdash V \textbf{ checks } v}{\Gamma\,[\Sigma]\vdash d\ v \textbf{ infers } C}$$

$$\frac{(\Gamma, \overline{f : \forall \overline{Z}.Q} \vdash e_i \textbf{ checks } Q_i)_i \qquad \Gamma, \overline{f : \forall \overline{Z}.Q}\,[\Sigma]\vdash d \textbf{ infers } C}{\Gamma\,[\Sigma]\vdash \mathsf{letrec}\ \overline{f : \forall \overline{Z}.Q = e}\ \mathsf{in}\ d \textbf{ infers } C}$$

$\boxed{\Gamma \vdash C \textbf{ checks } e}$

$$\frac{\Gamma\,[\Sigma]\vdash V \textbf{ checks } v}{\Gamma \vdash [\Sigma]V \textbf{ checks } v} \qquad \frac{\Gamma, x : V \vdash C \textbf{ checks } e}{\Gamma \vdash V \to C \textbf{ checks } \lambda x.e}$$

$$\frac{\Gamma\,[\Sigma]\vdash u \textbf{ infers } D\ \overline{U} \qquad C \textbf{ does } \Sigma \qquad (\Gamma, \overline{x_k} : \overline{V} \vdash C \textbf{ checks } e_k)_{k\ \overline{V} \in D\ \overline{U}}}{\Gamma \vdash C \textbf{ checks } \mathsf{case}\ u\ \mathsf{of}\ (k\ \overline{x_k} \mapsto e_k)_k}$$

$$\frac{\begin{array}{c}\Sigma' \text{ closed} \qquad C \textbf{ does } \Sigma \\ \Gamma\,[\Sigma \oplus \Sigma']\vdash d \textbf{ infers } [\Sigma \oplus \Sigma']V \qquad (\Gamma, \overline{x_c} : \overline{V}, g_c : \{U \to [\Sigma \oplus \Sigma']V\} \vdash C \textbf{ checks } e_c)_{c:U(\overline{V}) \in \Sigma'} \qquad \Gamma, x : V \vdash C \textbf{ checks } e\end{array}}{\Gamma \vdash C \textbf{ checks } \mathsf{handle}\ d\ \mathsf{with}\ (c\ \overline{x_c}\ ?\ g_c \mapsto e_c)_c \mid x \mapsto e}$$

**Figure 4.** Core Frank Typing Rules

In Frank, pattern matching trees $M$ are built up from leaves, case analysis, and handlers.

$$\begin{array}{rl} M ::= & \overline{e} \\ \mid & \mathsf{case}\ x\ \mathsf{of}\ (k\ \overline{x_k} \mapsto M_k)_{k \in D} \\ \mid & \mathsf{handle}\ x!\ \mathsf{with}\ (c\ \overline{x_c}\ ?\ g_c \mapsto M_c)_{c \in \Sigma} \mid x \mapsto M \end{array}$$

The leaves consist of a sequence of checkable computation expressions. Each element corresponds to one way of matching all of the patterns. If there exists a leaf with no elements, then the pattern matching is incomplete; if there exists a leaf with multiple elements, then the pattern matching is ambiguous. Our default strategy (as indicated by the thunk introduction rule) is to class incomplete pattern matching as a type error, and to keep only the first element in the case of ambiguous pattern matching. Our example generates the following pattern matching tree:

$$\begin{array}{l} M = \mathsf{handle}\ x_0!\ \mathsf{with} \\ \quad \mathit{send}\ x\ ?\ s \mapsto \\ \quad\quad \mathsf{handle}\ x_1!\ \mathsf{with} \\ \quad\quad\quad \mathit{receive}\ ?\ r \mapsto e_1 \\ \quad\quad\quad z \mapsto e_2 \\ \quad \mathit{abort}\ ?\ s \mapsto \\ \quad\quad \mathsf{handle}\ x_1!\ \mathsf{with} \\ \quad\quad\quad \mathit{receive}\ ?\ r \mapsto \mathit{bind}\ (r!\ \mathit{receive}!)\ e_3 \\ \quad\quad\quad z \mapsto \mathit{bind}\ z\ e_3 \\ \quad y \mapsto \\ \quad\quad \mathsf{case}\ y\ \mathsf{of} \\ \quad\quad\quad \mathit{unit} \mapsto \mathsf{handle}\ x_1!\ \mathsf{with} \\ \quad\quad\quad\quad \mathit{receive}\ ?\ r \mapsto \mathit{bind}\ (r!\ \mathit{receive}!)\ e_4 \\ \quad\quad\quad\quad z \mapsto \mathit{bind}\ z\ e_4 \end{array}$$

Each thunk pattern has been expanded out to explicitly list all of the cases according to its type. We obtain the corresponding Core

Frank code by abstracting over the fresh variables.

$$\lambda x_0\ x_1.M$$

Some pattern matching operations reorder columns as an optimisation. Column reordering is not in general a valid optimisation in Frank. This is because commands in the ambient effects, but not in the argument effects, are implicitly forwarded, and the order in which they are forwarded is left-to-right. (The forwarding behaviour is made precise in the Section 6.)

Of course, because Core Frank takes values as arguments whereas Frank takes computations, each argument must be wrapped in a thunk constructor. The type translation is given simply by the homomorphic extension of the following equation on function types:

$$(\!|[\Sigma]V|\!) \to (\!|C|\!) = \{[(\!|\Sigma|\!)](\!|V|\!)\} \to (\!|C|\!)$$

A correct pattern matching translation $(\!|-|\!)$ from Frank to Core Frank should be type preserving.

- If $\Gamma\ [\Sigma]\vdash\ u$ **infers** $V$ then $(\!|\Gamma|\!)\ [(\!|\Sigma|\!)]\vdash\ (\!|u|\!)$ **infers** $(\!|V|\!)$.
- If $\Gamma\ [\Sigma]\vdash\ V$ **checks** $v$ then $(\!|\Gamma|\!)\ [(\!|\Sigma|\!)]\vdash\ (\!|V|\!)$ **checks** $(\!|v|\!)$.
- If $\Gamma\ [\Sigma]\vdash\ d$ **infers** $C$ then $(\!|\Gamma|\!)\ [(\!|\Sigma|\!)]\vdash\ (\!|d|\!)$ **infers** $(\!|C|\!)$.

### 4.1 Incomplete and Ambiguous Pattern Matching as Effects

As an extension to Frank, we might allow incomplete and ambiguous pattern matching. The former may be permitted if the ambient effects contain the $Abort$ signature, in which case incomplete patterns are translated into the $abort : Zero$ command, which can then be handled however the programmer wishes. Similarly, we can define a $choice : X(X, X)$ command, in order to allow ambiguous pattern matches to be handled by the programmer.

## 5. Explicit Control Flow and Polymorphism

In order to make control flow explicit, and to ease the definition of an operational semantics, we translate Core Frank into a call-by-push-value calculus $F_{\text{eff}}$ based on Kammar et al's $\lambda_{\text{eff}}$ [8].

At the same time, we make polymorphism explicit. The syntax of $F_{\text{eff}}$ is given in Figure 5. Broadly, $F_{\text{eff}}$ types are similar to Frank types, with an explicit division between value type and computation types. A superficial difference is that rather than annotating value returning computations with effects, we shift such labels to the thunk containing the computation. The former design seems more convenient to program with, which is why we adopt it in the source language. The latter design leads to a more uniform presentation of the typing rules, and matches the design of $\lambda_{\text{eff}}$. Another minor difference is that in $F_{\text{eff}}$ commands must always be fully applied, which leads to a cleaner operational semantics. Note that let is actually redundant in $F_{\text{eff}}$, just as it is in Frank and Core Frank. We include let as a special construct because doing so makes the semantics cleaner.

The typing rules for $F_{\text{eff}}$ are given in Figure 6. Typing in $F_{\text{eff}}$ is unidirectional.

Call-by-push-value calculi such as $\lambda_{\text{eff}}$ and $F_{\text{eff}}$ make a strict separation between values and computations, not dissimilar from CPS or A-normal form representations, in which all reduction takes place at the level of computations. In such a setting it would seem most natural to add type abstractions to computations rather than values. However, this does not give us what we need in the presence of effects, as we need to be able to quantify over effects. Our solution is to build the universal quantifier into the thunk type (the introduction rule for thunks is the place where ambient effects are reified in a type) and build type application into forcing.

Types

$$\begin{array}{lll}
\text{(values)} & & U, V ::= D\ \overline{U} \mid \forall \overline{Z}.[\Sigma]\{C\} \mid X \\
\text{(computations)} & & C ::= \langle V \rangle \mid V \to C \\
\text{(quantifiers)} & & Z ::= X \mid \varepsilon \\
\text{(arguments)} & & T ::= V \mid \Sigma \\
\text{(polytypes)} & & P ::= \forall \overline{Z}.Q \\
\text{(monothunks)} & & Q ::= [\Sigma]\{C\} \\
\text{(signatures)} & & S ::= \cdot \mid c : U(\overline{V}), S \\
\text{(effects)} & & \Sigma ::= \varnothing \mid \Sigma, S\ \overline{V} \mid \varepsilon \\
\text{(type environments)} & & \Gamma ::= \cdot \mid \Gamma, x : V
\end{array}$$

Terms

$$\begin{array}{lll}
\text{(values)} & u, v ::= & x \mid k\ \overline{v} \mid \Lambda \overline{Z}.\{d\} \\
\text{(computations)} & d, e ::= & \mathsf{case}\ u\ \mathsf{of}\ (k\ \overline{x_k} \mapsto e_k)_k \mid (u\ \overline{T})! \\
& & \mid\ \lambda x.e \mid d\ v \mid \mathsf{ret}\ v \mid \mathsf{let}\ x = e\ \mathsf{in}\ e' \mid c\ \overline{v} \\
& & \mid\ \mathsf{handle}\ d\ \mathsf{with}\ (c\ \overline{x_c}\ ?\ g \mapsto e_c)_c \mid x \mapsto e \\
& & \mid\ \mathsf{letrec}\ \overline{f : P = e}\ \mathsf{in}\ e'
\end{array}$$

**Figure 5.** $F_{\text{eff}}$ Syntax

The type translation from Core Frank to $F_{\text{eff}}$ is given by the homomorphic extension of the following equations:

$$\begin{array}{rl}
[\![\{C\}]\!] & = [[\![\Sigma]\!]]\{[\![C]\!]\}, \quad \text{where } C\ \mathbf{does}\ \Sigma \\
[\![[\Sigma]V \to C]\!] & = [[\![\Sigma]\!]][\![V]\!] \to [\![C]\!] \\
[\![[\Sigma]V]\!] & = \langle [\![V]\!] \rangle
\end{array}$$

In order to simplify the definition of the term translation, as a preprocessing step we annotate instances of polymorphic variables $f$ with the type arguments they are instantiated with and commands with their arities. The term translation is a call by value embedding of Core Frank into $F_{\text{eff}}$:

$$\begin{array}{rl}
[\![x]\!] & = \mathsf{ret}\ x \\
[\![f^{\overline{T}}]\!] & = \mathsf{ret}\ \{f\ [\![\overline{T}]\!]\} \\
[\![c^n]\!] & = \mathsf{ret}\ \{\lambda x_1 \ldots x_n.c\ x_1\ \ldots\ x_n\} \\
\\
[\![k\ \overline{v}]\!] & = \mathsf{let}\ \overline{x} = [\![\overline{v}]\!]\ \mathsf{in}\ \mathsf{ret}\ (k\ \overline{x}) \\
[\![\{e\}]\!] & = \mathsf{ret}\ \{[\![e]\!]\} \\
\\
[\![d\ v]\!] & = \mathsf{let}\ x = [\![v]\!]\ \mathsf{in}\ [\![d]\!]\ x \\
[\![u!]\!] & = \mathsf{let}\ x = [\![u]\!]\ \mathsf{in}\ x! \\
[\![\mathsf{letrec}\ \overline{f : \forall \overline{Z}.Q = e}\ \mathsf{in}\ d]\!] & = \mathsf{letrec}\ \overline{f : [\![\forall \overline{Z}.Q]\!] = [\![e]\!]}\ \mathsf{in}\ [\![d]\!]
\end{array}$$

$$\begin{array}{rl}
[\![\lambda x.e]\!] & = \lambda x.[\![e]\!] \\
[\![\mathsf{case}\ u\ \mathsf{of}\ (k\ \overline{x_k} \mapsto e_k)_k]\!] & = \\
\quad \mathsf{let}\ x = [\![u]\!]\ \mathsf{in}\ \mathsf{case}\ x\ \mathsf{of}\ (k\ \overline{x_k} \mapsto [\![e_k]\!])_k & \\
[\![\mathsf{handle}\ d\ \mathsf{with}\ (c\ \overline{x_c}\ ?\ g_c \mapsto e_c)_c \mid x \mapsto e]\!] & = \\
\quad \mathsf{handle}\ [\![d]\!]\ \mathsf{with}\ (c\ \overline{x_c}\ ?\ g_c \mapsto [\![e_c]\!])_c \mid x \mapsto [\![e]\!] &
\end{array}$$

PROPOSITION 1. *The translation $[\![-]\!]$ from Core Frank to $F_{\text{eff}}$ is type preserving.*

- *If* $\Gamma\ [\Sigma]\vdash\ u$ **infers** $V$ *then* $[\![\Gamma]\!]\ [[\![\Sigma]\!]]\vdash\ [\![u]\!] : \langle [\![V]\!] \rangle$.
- *If* $\Gamma\ [\Sigma]\vdash\ V$ **checks** $v$ *then* $[\![\Gamma]\!]\ [[\![\Sigma]\!]]\vdash\ [\![v]\!] : \langle [\![V]\!] \rangle$.
- *If* $\Gamma\ [\Sigma]\vdash\ d$ **infers** $C$ *then* $[\![\Gamma]\!]\ [[\![\Sigma]\!]]\vdash\ [\![d]\!] : [\![C]\!]$.
- *If* $\Gamma \vdash C$ **checks** $e$ *and* $C$ **does** $\Sigma$ *then* $[\![\Gamma]\!]\ [[\![\Sigma]\!]]\vdash\ [\![e]\!] : [\![C]\!]$.

Kammar et al [8] classify a number of different varieties of handler. The handlers in $F_{\text{eff}}$ (and indeed Frank) are *shallow* in that, unlike Plotkin and Pretnar's original deep handlers, the handler is not automatically rewrapped around the continuation when handling a command. Roughly, deep handlers perform a fold over computa-

$\boxed{\Gamma \vdash v : V}$

$$\frac{x : V \in \Gamma}{\Gamma \vdash x : V} \qquad \frac{\Gamma \vdash V_i : v_i \qquad k\ \overline{V} \in D\ \overline{U}}{\Gamma \vdash D\ \overline{U} : k\ \overline{v}} \qquad \frac{\Gamma\ [\Sigma]\!\vdash e : C}{\Gamma \vdash \Lambda\overline{Z}.\{e\} : \forall\overline{Z}.[\Sigma]\{C\}}$$

$\boxed{\Gamma\ [\Sigma]\!\vdash e : C}$

$$\frac{\Gamma \vdash u : D\ \overline{U} \qquad (\Gamma, \overline{x_k} : \overline{V}\ [\Sigma]\!\vdash e_k : C)_{k\ \overline{V} \in D\ \overline{U}}}{\Gamma\ [\Sigma]\!\vdash \mathsf{case}\ u\ \mathsf{of}\ (k\ \overline{x_k} \mapsto e_k)_k : C} \qquad \frac{\Gamma \vdash u : \forall\overline{Z}.[\Sigma]\{C\}}{\Gamma\ [\Sigma]\!\vdash (u\ \overline{T})! : C[\overline{T}/\overline{Z}]} \qquad \frac{\Gamma, x : V\ [\Sigma]\!\vdash e : C}{\Gamma\ [\Sigma]\!\vdash \lambda x.e : V \to C}$$

$$\frac{\Gamma\ [\Sigma]\!\vdash d : V \to C \qquad \Gamma \vdash v : V}{\Gamma\ [\Sigma]\!\vdash d\ v : C} \qquad \frac{\Gamma \vdash v : V}{\Gamma\ [\Sigma]\!\vdash \mathsf{ret}\ v : \langle V \rangle} \qquad \frac{\Gamma\ [\Sigma]\!\vdash d : \langle V \rangle \qquad \Gamma, x : V\ [\Sigma]\!\vdash e : C}{\Gamma\ [\Sigma]\!\vdash \mathsf{let}\ x = d\ \mathsf{in}\ e : C}$$

$$\frac{c : U(\overline{V}) \in \Sigma \qquad \Gamma \vdash v_i : V_i}{\Gamma\ [\Sigma]\!\vdash c\ \overline{v} : \langle U \rangle}$$

$$\frac{\Gamma\ [\Sigma \oplus \Sigma']\!\vdash d : \langle V \rangle \qquad (\Gamma, \overline{x_c} : \overline{V}, g : [\Sigma \oplus \Sigma']\{U \to \langle V \rangle\}\ [\Sigma]\!\vdash e_c : C)_{c : U(\overline{V}) \in \Sigma'} \qquad \Gamma, x : V\ [\Sigma]\!\vdash e : C}{\Gamma\ [\Sigma]\!\vdash \mathsf{handle}\ d\ \mathsf{with}\ (c\ \overline{x_c}\ ?\ g \mapsto e_c)_c \mid x \mapsto e : C}$$

$$\frac{(\Gamma, \overline{f : \forall\overline{Z}.Q}\ [\Sigma]\!\vdash e_i : Q_i)_i \qquad \Gamma, \overline{f : \forall\overline{Z}.Q}\ [\Sigma]\!\vdash d : C}{\Gamma\ [\Sigma]\!\vdash \mathsf{letrec}\ \overline{f : \forall\overline{Z}.Q = e}\ \mathsf{in}\ d : C}$$

**Figure 6.** $F_{\text{eff}}$ Typing rules

tions, whereas shallow handles perform a case split. Deep handlers are denotationally better behaved than shallow handlers, but shallow handlers sometimes appear more convenient to program with. The handlers in $F_{\text{eff}}$ (and indeed Frank) are polymorphic forwarding handlers. The form of polymorphism amounts to a kind of row polymorphism with shadowing, unlike the more conventional kind of row polymorphism suggested by Kammar et al.

## 6. Small-step semantics

We give a small step operation semantics for $F_{\text{eff}}$ inspired by Kammar et al's semantics for $\lambda_{\text{eff}}$ [8]. All of the rules except the ones for handlers are pretty standard $\beta$-reductions. The rules for forcing are slightly unusual due to the hard-wiring of polymorphism into thunks.

Returns $\mathsf{ret}\ v$ 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. If a clause is defined in the handler, then that clause is selected. If not, then the command is forwarded to be handled by an outer handler, but its continuation is handled by the current handler.

Delimited computation contexts $D$ are used to characterise the continuation up to the current handler. Computation contexts $E$ amount to evaluation contexts for a call-by-push-value setting.

PROPOSITION 2 (Type Soundness).

- *If* $\Gamma\ [\Sigma]\!\vdash d : C$ *and* $d \longrightarrow e$ *then* $\Gamma\ [\Sigma]\!\vdash e : C$.
- *If* $\cdot\ [\varnothing]\!\vdash d : \langle V \rangle$ *then either there exists* $v$ *such that* $d = \mathsf{ret}\ v$ *or there exists* $e$ *such that* $d \longrightarrow e$.

## 7. Related Work

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

A natural implementation for handlers is to use *free monads* [8]. Swierstra [27] illustrates how to write effectful programs with free monads in Haskell, taking advantage of type-classes to provide a certain amount of modularity.

Inspired by Bauer and Pretnar's Eff, Visscher has implemented the `effects` library [28]. The key idea is to layer continuation monads in the style of Filinski [5], using Haskell type classes to automatically infer lifting between layers.

Filinski's work on monadic reflection and layered monads is closely related to effect handlers [6]. 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

Languages other than Frank that attempt to elide some effect variables from source code include Links [12] and Daan Leijen's Koka [10]. Neither eliminates effect variables altogether.

Swamy et al [26] add support for monads in ML, supporting direct-style effectful programming in a strict language. Unlike Frank, their system is based on monad transformers rather than effect handlers.

## 8. Future Work

Our first priority is to implement a prototype for the current Frank design. Having done that, there is much scope for exploring different implementations of handlers, both for performance and for exploring new abstractions. We would like to combine effect handlers with richer type systems, following the work of McBride [15] and Brady [4]. Handlers provide some of the same functionality as modules and type classes. We would like to formally relate all three abstractions. We are exploring algebraic effects and effect handlers for idiom [17] and arrow [7] computations.

## References


[1] L. Augustsson. Compiling pattern matching. In *FPCA*, pages 368–381, 1985.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

$$
\begin{array}{rl}
\text{(Delimited computation contexts)} & D ::= [\,] \mid D\ v \mid \mathsf{let}\ x = D\ \mathsf{in}\ e \\
\text{(Computation contexts)} & E ::= [\,] \mid E\ v \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{handle}\ E\ \mathsf{with}\ H
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{case}\ k_j\ \overline{v}\ \mathsf{of}\ (k_i\ \overline{x_i} \mapsto e_i)_i & \longrightarrow e_j[\overline{v}/\overline{x_j}] \\
((\Lambda \overline{Z}.\{e\})\ \overline{T})! & \longrightarrow e[\overline{T}/\overline{Z}] \\
(\lambda x.e)\ v & \longrightarrow e[v/x] \\
\mathsf{let}\ x = \mathsf{ret}\ v\ \mathsf{in}\ e & \longrightarrow e[v/x] \\
\mathsf{letrec}\ \overline{f : \forall \overline{Z}.Q = e}\ \mathsf{in}\ d & \longrightarrow d[\overline{\Lambda Z.\{e\}}[\Lambda Z.\{\mathsf{letrec}\ \overline{f : \forall \overline{Z}.Q = e}\ \mathsf{in}\ (f\ \overline{Z})!\}/\overline{f}]/\overline{f}] \\
\mathsf{handle}\ (\mathsf{ret}\ v)\ \mathsf{with}\ H & \longrightarrow H(\mathsf{ret}, v) \\
\mathsf{handle}\ D[c\ \overline{v}]\ \mathsf{with}\ H & \longrightarrow H(c, \overline{v}, \{\lambda z.D[\mathsf{ret}\ z]\})
\end{array}
$$

where the action of $H = (c_i\ \overline{x_i}\ ?\ g \mapsto e_i)_i \mid x \mapsto e$ is given by:

$$
\begin{array}{l}
H(\mathsf{ret}, v) = e[v/x] \\
H(\mathsf{c_j}, \overline{v}, u) = e_j[\overline{v}/\overline{x}, u/g] \\
H(\mathsf{c}, \overline{v}, u) = \mathsf{let}\ z = c\ \overline{v}\ \mathsf{in}\ \mathsf{handle}\ u!\ z\ \mathsf{with}\ H, \quad c \neq c_i \text{ for any } i
\end{array}
$$

$$
\frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}
$$

**Figure 7.** Small-step operational semantics for $F_{\text{eff}}$


[3] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. In *CALCO*, volume 8089 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[4] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*. ACM, 2013.

[5] A. Filinski. Representing layered monads. In *POPL*. ACM, 1999.

[6] A. Filinski. Monads in action. In *POPL*. ACM, 2010.

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

[8] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *ICFP*, pages 145–158. ACM, 2013.

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

[10] D. Leijen. Koka: Programming with row-polymorphic effect types. Technical Report MSR-TR-2013-79, Microsoft Research, 2013.

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

[12] S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*. ACM, 2012.

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

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

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

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

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

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

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

[20] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In *FoSSaCS*. Springer-Verlag, 2001.

[21] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS*. Springer-Verlag, 2002.

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

[23] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP*. Springer-Verlag, 2009.

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

[25] M. Pretnar. Inferring algebraic effects. *CoRR*, abs/1312.2334, 2013.

[26] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP*. ACM, 2011.

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

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