School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Linear Frank

by

James Treloar

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted:	December 2020	Student ID:	z5112793
Supervisor:	Dr. Christine Rizkallah	Topic ID:	

This thesis explores the extension of Frank with the linear types. Frank is a functional programming language built around the algebraic effects and effect handlers abstraction. Algebraic effects structure computational effects (state, exceptions, non-determinism, I/O, concurrency) as collections of abstract operations. Handlers give interpretations to these abstract operations. However, without knowing whether a given side effect will not continue, continue once or continue many times, it becomes difficult to reason about fragile effects such as file I/O. Linear types are an incremental step towards allowing the creation of handlers that provide a way of distinguishing and enforcing the number of times an effect may continue.

Contents

1 Introduction. **1**

2 Background. **4**

2.1 Linear types. 4

2.2 Algebraic effects and handlers. 4

3 Linear Frank. **7**

4 A Linear Frank formalism. **11**

4.1 Linear type system 12

5 Related and future work **17**

5.1 Runners. 17

5.2 Deep finalization. 18

6 Analysis of the results. **20**

7 Conclusion **22**

Bibliography **23**

Chapter 1

Introduction.

Computational effects, such as file I/O, mutable state, non-determinism and exception handling, are the constructs by which a program can be structured with in order to perform useful work. Arbitrary use of computational effects can lead to programs that violate certain safety properties; unrestricted memory allocations can lead to memory leaks, inconsistent use of file handles can lead to I/O errors and process forking can lead to concurrency issues. To solve this issue programming language research has focused on ways to structure how these effects are represented, thereby limiting the ability of a programmer to introduce errant behaviour that could otherwise be statically avoided.

One of the more historical mechanisms to structure effects is to use monads[Jon03], which sequence instructions for an effect and provide fixed interpretations for the semantics of that effect. This thesis does not focus on monads but instead has delved into the realm of algebraic effects and their handlers. This is another way to structure computational effects that splits them into abstract algebraic effects, which form the interface of an effect, and effect handlers, which provides specific interpretations to the algebraic effect. The basic idea is that when a computation with access to a set of commands from an algebraic effect is computed, it will resolve into a value or an command that can be interpreted by a supplied effect handler.

Frank is a language built around the algebraic effects and effect handler abstraction. Frank combines functions and effect handlers into the same construct, called an operator. Operators can adjust the effects that can be used in the evaluation of a particular argument and can provide interpretations of those effects when used in the evaluation of the argument. Similarly, an operator can require a certain set of effects to have available interpretations to be able to invoke an operator.

An example type signature of an operator that uses commands from the `Abort` effect and provides the `File` effect on its first argument could look like `op : { (<File>X) [Abort] X }`. This operator, called `op`, will take a single argument that eventually will turn into something with the polymorphic type `X`. Commands that belong to the `File` effect may be used within this first argument, and the interpretation will be implemented in the definition of the operator. Furthermore, the operator itself can use the commands in the `Abort` effect, and this ability is ambiently pushed down into the evaluation of the first argument as well, reducing the need to manually specify how effects are pushed around a program.

To provide an interpretation of commands, handlers are provided a resumption, a value much like a continuation which when invoked will continue program evaluation from after the point where the command was invoked. The problem dealt with in this thesis then lies in determining how many times this resumption is allowed to be used. If it is never used, then the remaining computation will never be run. If it is used many times, then the remaining computation will be run many times. As the interpretation is abstracted away from the programmer and the exact behaviour of an effect is not transparent, it is not clear how the program might behave.

```
let fh = open "file.txt" in
let line = read fh in
    tell line;
    close fh
```

Figure 1.1: Sample code where the evaluation depends on command interpretations.

To illustrate this point, we lay out a simple program which uses two effects, a file effect and a writer effect. The file effect provides four commands, open, close, write and read. The writer effect provides a single command, which is tell. As the exact interpretations of these commands are abstracted away, it is impossible to say which parts of the computation will actually run. If the interpretation for tell has the capability of not continuing, then the file handle may never be closed.

To solve this issue we have implemented linear types into the Frank language. Linear types are used to represent values that may be used exactly once. By providing the static ability to check that a given value is only ever given once, the tool is given to the programmer to make safer guarantees about the handlers that are provided to interpret their algebraic effects. To the best of our knowledge, linear types and algebraic effect systems have not been investigated before, and exploring the usage of linear types in an algebraically effectful world is the purpose of this thesis.

The primary goal of this thesis has been to bring static guarantees to dealing with resources that need a certain ordering of effects to be used, such as file I/O. While this has not been fully realized, in the setting of algebraic effects bringing linear types into the fray seems to be a necessary step in achieving the ideal of even safer effects.

In this thesis we will:

- Introduce the concept of linear types, and provide a brief history and motivation for their use.
- Explain algebraic effects and their handlers in greater detail, and give context to Frank by discussing some of its peers.
- Present a Linear Frank tutorial, to demonstrate the programming model for which algebraic effects, effect handlers and linear types allow.

- Outline the changes to Frank's typing rules that allow and track the usage of values.
- Assert a handful of properties on the nature of Frank's type rules, as well as the guarantees given from linearity.
- Discuss the use of linear types as the solution to the unknown continuation problem.
- Touch upon the relevant related work, and propose future avenues of interest in regards to this topic.

Chapter 2

Background.

2.1 Linear types.

In any standard modern programming language we would not blink twice at a function that takes a number, adds it together with itself and returns the result. Generally we are allowed to duplicate or drop values as much as we care to, but this flexibility with values is often not desired. Sometimes we wish to model value in our programming languages which correspond to things that can only be used once, resources that cannot be duplicated arbitrarily and cannot be readily discarded.

To be able to introduce statically checked values that can only be used a specific amount of times, linear types can be used. The type systems underpinning linear types come in many flavours, including Wadler's extension of intuitionistic logic into linear logic[Wad90], Turner & Wadler's more explicit usage counting[onc], Cogent's uniqueness typing[AHC$^+$16], Morris' qualified types for linear functions[Mor16] and Orchard's graded modal types[OLEI19].

To be able to introduce some sort of linear quality to the handlers in Frank, inspiration was taken from Morris'[Mor16] system of annotating function types with linearity, as opposed to the concrete values themselves. This lends itself to a quality of the bidirectional type system that Frank uses.

2.2 Algebraic effects and handlers.

Algebraic effects[PP03] and their corresponding handlers[PP13] are a relatively new concept in the domain of programming language research. Together, they provide a mechanism for implementing novel control-flow logic including threaded implicit state, mutable state, ref types, exception handling and non-determinism. While these computational effects may not be particularly exciting by themselves, by creating languages that treat effects and their handlers as first class citizens, these computational effects can be provided from within the language itself and does not need to be hardcoded into the language's semantics. This affords the flexibility to be able to create new computational effects without needing to incorporate them into the languages runtime.

Algebraic effects and their handlers are not the only way to do this, another common way to create new computational effects is to use monads. However, algebraic effects and their handlers bring the additional flexibility of separating the computational effect into an interface containing the abstract commands of an effect, and the actual interpretations given to the commands through handlers. The advantage of this is that code can be written in terms of an abstract algebraic effect, but does not need to specify which handler should be used, allowing different handlers to be swapped and interchanged seamlessly.

The `writeToFile` example below is of a Linear Frank operator, and while the syntax will be discussed in more detail in the next section, the basic premise is that it accepts a filename and a list of strings, and will open a file, apply the write command to each lines in the list and close the file. The three abstract commands belonging to the file effect here `open`, `write` and `close`.

```
writeToFile filename lines =
 open filename; map write lines; close!
```

Figure 2.1: Operator that will accept a filename, open the file, write the contents of the list to the file and subsequently close the file.

The actual runtime effect of running this code is completely dependent on the concrete interpretations of the file effect that is supplied. The most straightforward idea would be to provide at the runtime level operating system primitives that properly open and close a file, and where writes immediately write to disc. Alternatively one could easily provide a new handler for these commands that caches writes and perform the actual write on file close, perhaps for performance reasons. Finally, another way would be to enable proper unit testing of segments of code by replacing the handler with one that does not actually write to a file. All this is possible to do without having to change the actual logic.

Frank is not the only language to support in its design a mechanism for specifying and implementing algebraic effects and their handlers. There exists many languages or adaptations to existing languages[Lei14, BP15, Hil] that afford this flexibility of defining computational effects, but Frank does provide strength in a few dimensions. The first aspect is that functions and handlers have been unified into a single concept, the operator. Operators use effects at the same time as providing interpretations to further effects. This makes writing complex handlers, such as multi-handlers, where multiple effects can be provided to different areas of computation and interleaved between themselves. Other languages providing handlers, Koka being one example, introduce handlers as a separate construct on top of regular functions.

Another differentiating aspect of Frank is that Frank's handlers are shallow[HL18] as opposed to deep. A deep handler's continuation continues to use the same handler, using an implicit recursion to continue computation. A shallow handler's continuation is not implicitly handled, meaning any recursion has to be made explicit. At the cost of verbosity (as the recursion always has to be made explicit) shallow handlers allow an easier starting place from which to change handlers on the fly and to allow multi-handlers. Shallow handlers can represent the same programs that deep handlers can by always recursively calling the same handler. Deep handlers can represent shallow handlers by mutually recursive data types as demonstrated by Hillerstrom and Lindley[KLO13].

Chapter 3

Linear Frank.

Frank is a bidirectionally typed, directly applied functional programming language that places an emphasis on the mechanisms by which effects are tracked. Frank conflates the two notions of functions and effect handlers into a single concept called an operator, and streamlines the way by which effectful programming can be performed. A simple Frank operator that divides a list of integers by a denominator might look as follows.

```
divide : { (List Int, Int)  [Abort] List Int }
divide _ 0 = abort!
divide [] _ = []
divide (x :: xs) denom = x / denom :: xs
```

Figure 3.1: A operator defining a safe division operator relying on an abort computational effect for error handling.

There is a lot to unpack here, first and foremost the strange syntax of the operator. An operator consists of two major components, a type signature and a list of clauses, providing pattern matches onto the arguments. The type signature of the operator consists of a suspended computation (a computation that is yet to run) which contains a number of arguments, a list of effects required by this operator to be able to be invoked and a result type. In this case the arguments are a list of integers, and an integer. The return type is a list of integers, and the operator requires a handler for the abort effect, which can be used to signal a failed computation, to be supplied.The operator clauses are pattern matched against the arguments from top to bottom, and the right hand side represents the computation to be evaluated after the patterns are matched. Of the three clauses, the first handles the case where division by zero is attempted and the computation is aborted, the second handles the base case for when there are no more elements left in the list and the last case performs the division and a recursion on the list.

To be able to provide the ability to use an effect, we need to be able define the abstract interface of a computational effect and be able to define an interpretation. As has been mentioned we are able to do so using an operator which captures the commands invoked as requests to be interpreted on each argument position. To illustrate how to encode a mutable state computational effect in Frank, we can define the abstract interface as two commands that let us retrieve and set the state, along with a handler to provide an interpretation.

```
interface State X = put : (X) Unit
                  | get : () X

execState : { (Y, <State Y>X) X }
execState state <get -> k> = execState state (k state)
execState _ <put state -> k> = execState state (k state)
execState _ result = result
```

Figure 3.2: Effect definition and handler for the state effect, providing the ability to perform mutably stateful computation.

We can define our algebraic effects as a number of operator signatures representing the abstract operations available to this effect. In this case we have a command to put an `X` into the state, and to retrieve an `X` from the state. To give a particular interpretation to this effect we define an operator `execState` which is provided with an initial state in the first argument, and then a second argument that has been adjusted with the ability to perform the effectful commands `get` and `put`. The type within the state effect is polymorphic, and will resolve to the same type as the given initial state. Most interesting at this point are the first and second clauses of `execState`, where the interpretation of each command is given by first capturing a request to interpret the command, the arguments supplied to the command, and a continuation representing the point immediately after where the command had been invoked.

In this handler we used the continuation supplied to each request only once, but that does not need to be the case. If we wanted to define a computational effect to signal a failure state, we could introduce an algebraic effect and handlers to do so. The definition of failure that we will provide is to return just a value if computation succeeds, or nothing if it does not, which can be modelled by a simple new data type. Note that in the handling of the `abort` command, the continuation is discarded and computation of the argument will halt.

```
interface Abort = abort X : X

data Maybe X = just X | nothing

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

Figure 3.3: Effect definition and handler for an abort computational effect, that allows a computation to stop computation when an errant state is enterred.

Similarly, there is nothing at the moment that would stop us using the continuation multiple times. If one wanted to collate all the possible results from a many branching piece of code, then this could be simulated by representing choice as a computational effect, and running both sides of each boolean choice.

```
interface Choice = choice : Bool

allChoices : { (<Choice>X) List X }
allChoices <choice -> k> = concat (allChoices (k true)) (allChoices (k false))
allChoices x = [x]
```

Figure 3.4: Effect definition and handler for a choice computational effect, that allows all the results of a non-deterministic choice to be evaluated and concatenated.

Then, given a suitable definition of `if`, we are able to non-deterministically collate all possible evaluations of a computation.

```
if : { (Bool, {X}, {X}) X }
if  true  body _ = body!
if false  _ body = body!

> allChoices ( let a = if choice! { 1 } { 2 } in
               let b = if choice! { 10 } { 20 } in
               a + b )
=> [11, 20, 12, 22]
```

Figure 3.5: Using the choice effect.

The number of times that an invoked command may continue is unknown unless usages are attached to the effects that are being handled. Linear types in Frank fall into two categories, a value will either be used exactly once or it may be used many times. Additionally, values which may be used many times can be coerced into values that may only be used. An identity operator to accomplish this would be rendered as follows.

```
once : { ( w: X ) 1: X }
once x = x
```

Figure 3.6: A trivial linear identity operator.

Type annotations in an operator are annotated to a type by prepending it with the desired usage, as in `p: X` where `p` is a usage and `X` is the type. The explicit linear usage is `1`, while the explicit many usage is `w`. It would be far too unwieldy if usages were required to be explicitly defined, and if this was the case then there would be no ability to write a flexible identity operator that would work with both linear and non-linear values. However, a method of usage polymorphism is implemented, where over the scope of an operator type definition, using the same free usage annotations will allow them to resolve during unification to the same usage. This allows us to write a general identity operator.

```
id : { (p : X) p : X }
id x = x
```

Figure 3.7: A universal identity operator.

As a matter of convention, and for the sake of readability, if the usage annotation for a type is omitted, then over the scope of an operator the usage annotation is fixed to the same value. This is how our earlier Frank snippets were able to be written without the need for specifying the concrete usages required.

The interleaving of usages in an operator's type can become quite confusing. The `map` operator requires a number of rather specific usages in order to be maximally flexible. In particular it is not possible to create non-linear data types that house linear types.

```
map : { ( w: { (r: X) t: Y }, q: List (r: X)) q: List (t: Y) }
map _ [] = []
map f (x::xs) = f x :: map f xs
```

Figure 3.8: Usage flexible map operator.

Breaking this down, the first argument to map has type `w: { (r: X) t: Y }`, which is an operator which can be used many times, and takes an argument `r: X` and returns a result of type `t: Y`. The second argument is `q: List (r: X)`, a list that has a usage of `q` and contains elements of type `r: X`. The result type is a list of type `q: List (t: Y)`. By judicious use of usage variables it is possible to define a map operator that can handle any permutation of usage variables in input while maintaining a correct output type.

Linear usages in Frank have also been extended to interact with the implementation of effect handlers. The usage of an argument also affects the usage of the continuation variable for the effects of any effect adjustments on that argument. This means that if we wanted to provide an effect handler for our state effect, we could implement it with a linear adjustment.

```
interface State X = put : (X) Unit
                          | get : () X

linearState : { (p: X, 1: <State (p: X)>Y) 1: Y }
linearState state <get -> k> = linearState state (k state)
linearState _ <put state -> k> = linearState state (k unit)
linearState _ x = x
```

Figure 3.9: A handler for the state effect which linearly sequences the state.

Chapter 4

A Linear Frank formalism.

This thesis, since diverging from attempting to directly provide a mechanism for controlling resourceful effects with runners[AB19], has instead fallen back to a more general approach in implementing linear types into the type system of the language such that linear effects can be explicitly formulated. This has involved modifying the existing Frank type system in such a way that linear types can be represented, then implementing that into the Frank reference implementation's type checker. Here we present the changes that were made to introduce linear types into the type system.

First we acknowledge what did not need to have changed from the original formulation of Frank. The well-formedness rules and actions on adjustments were left as is, as they were contingent on the introduction of linear types. A new set of changed operational semantics were not introduced, although ideally changes representing when and how linear values are introduced and eliminated would have been preferred. What was provided was a new surface syntax that allowed usages on types to be clearly annotated, and an adjustment of the typing rules that allow the introduction of linear types.

4.1 Linear type system

usage value	$\rho, \pi ::= 1, \omega, j$
data types	D
value type variables	X
computation types	$C ::= \overline{(\overset{\rho}{:} T)} \overset{\pi}{:} G$
effect type variables	E
value types	$A, B ::= D\,\overline{R} \mid \{C\} \mid X$
argument types	$T ::= \langle \Delta \rangle A$
return types	$G ::= [\Sigma] A$
type binders	$Z ::= X \mid [E]$
type arguments	$R ::= A \mid [\Sigma]$
polytypes	$P ::= \forall\, \overline{Z}.A$
interfaces	I
term variables	x, y, z, f
instance variables	s, a, b, c
seeds	$\sigma ::= \varnothing \mid E$
abilites	$\Sigma ::= \sigma \mid \Xi$
extensions	$\Xi ::= \iota \mid \Xi, I\,\overline{R}$
adaptors	$\Theta ::= \iota \mid \Theta, I(S \to S')$
adjustments	$\Delta ::= \Theta \Vert \Xi$
instance patterns	$S ::= s \mid S\ a$
kind environments	$\Phi, \Psi ::= \cdot \mid \Phi, Z$
type environments	$\Gamma ::= \cdot \mid \Gamma, x \overset{\rho}{:} A \mid \Gamma, f \overset{\rho}{:} P$
instance environments	$\Omega ::= s : \Sigma \mid \Omega, a : I\,\overline{R}$

Figure 4.1: Linear Frank types.

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

Figure 4.2: Linear Frank terms.

Frank as originally specified functions as a bidirectional type system[DK19]. A bidirectional type system offers two fundamental judgement forms, a checking form where all the items of information are supplied, and an inference form that will synthesize a type corresponding to the given form. The new linear type system follows the same structure as the original, but has instead been modified to track the relationship of the type usages, and to codify the flow of usage information.

The two forms used by the bidirectional type system are:

- $[\Sigma]\!\!- x \overset{\rho}{:} A$, which under kind environment Φ, variable context Γ and ambient ability Σ will check that the terms x has type A with usage ρ.
- $[\Sigma]\!\!- x \overset{\rho}{\Rightarrow} A$, which under kind environment Φ, variable context Γ and ambient ability Σ will synthesize from the terms x that the corresponding type is A with usage ρ.

Furthermore, often a sequence of terms or types are represented by an overbar or a subscripted parentheses, as in $\overline{x}$ or $(x_i)_i$.

$$\frac{}{\omega \leqslant 1} \qquad \frac{\rho = \omega}{\omega \leqslant \rho} \qquad \frac{\rho = 1}{\rho \leqslant 1}$$

Figure 4.3: Various rules describing the ordering of usage values.

The values of a usage can concretely be either 1 or ω (representing linear and unrestricted), or can range somewhere between.

$$\frac{\Phi;\Gamma\,[\Sigma]\!\!-\ m \overset{\rho}{\Rightarrow} A \qquad A = B \qquad \rho \leqslant \pi}{\Phi;\Gamma[\Sigma]\!\!-\ \downarrow m \overset{\pi}{:} B}\text{T-Switch}$$

Figure 4.4: T-Switch type rule.

For the purposes of usage checking, the heavy lifting is primarily performed by the side condition in the T-Switch rule. The purpose of this rule is to provide a boundary to check a term when the type is not immediately available, and has to be inferred. Along with the check that both the supplied checking type and the subsequently inferred type are equal, there is an assertion that the inferred type must be as or less permissive than the supplied checking type. This means if a term is being checked with a linear type the inferred type can only be as permissive, and must be linear. If the supplied usage is unrestricted, then the inferred type can either be as permissive, which would also be unrestricted, or it may be linear, which is less permissive.

$$\dfrac{\dfrac{\Sigma' = \Sigma \qquad (\Sigma \vdash \Delta_i \dashv \Sigma'_i)_i}{\Phi; \Gamma\,[\Sigma]\!\vdash\, m \overset{\xi}{\Rightarrow} \{\overline{(\overset{\rho}{:} \langle\Delta\rangle A)}\ \overset{\pi}{:}\,[\Sigma']B\}} \qquad (\Phi; \Gamma'_i\,[\Sigma'_i]\!\vdash\, n_i \overset{\rho_i}{:} A_i)_i}{\Phi; \Gamma, \overline{\Gamma'}\,[\Sigma]\!\vdash\, m\,\overline{n} \overset{\pi}{\Rightarrow} B}\text{T-App}$$

$$\dfrac{k\,\overline{(\overset{\rho}{:} A)} \in D\,\overline{R} \qquad (\Phi; \Gamma_i\,[\Sigma]\!\vdash\, n_i \overset{\rho_i}{:} A_i)_i \qquad (\rho_i \leqslant \pi)_i}{\Phi; \overline{\Gamma}\,[\Sigma]\!\vdash\, k\,\overline{n} \overset{\pi}{:} D\,\overline{R}}\text{T-Data}$$

$$\dfrac{\Phi \vdash \overline{R} \qquad c : \forall\overline{Z}.(\overline{(\overset{\rho_j}{:} A)}\ \overset{\pi}{:} B) \in \Sigma \qquad (\Phi; \Gamma_j\,[\Sigma]\!\vdash\, n_j \overset{\rho_j}{:} A_j[\overline{R}/\overline{Z}])_j}{\Phi; \overline{\Gamma}\,[\Sigma]\!\vdash\, c\,\overline{R}\,\overline{n} \overset{\pi}{:} B[\overline{R}/\overline{Z}]}\text{T-Command}$$

Figure 4.5: Application and application-like rules.

The core rule where a lot of the usage information is mediated is the application rule (T-App) which given an operator an a list of arguments will synthesize a type for the operator, and check each of the operands against the supplied arguments, ensuring that the operator supplied actually matches the arguments supplied. Should this be the case, the rule will infer the return type of the inferred operator as the output type. The two other rules that act similarly to application (T-Data and T-Command) do not need to infer a type as the type of the operator is discovered through mechanisms where the calling context will already know the type. Command types are tracked in the ambient ability and data constructors are tracked in the kind environment.

$$\dfrac{}{\Phi; x \overset{\rho}{:} A + \omega\Gamma\,[\Sigma]\!\vdash\, x \overset{\rho}{\Rightarrow} A}\text{T-Var}$$

$$\dfrac{\Phi \vdash \overline{R}}{\Phi; f \overset{\rho}{:} \forall Z.A + \omega\Gamma\,[\Sigma]\!\vdash\, f\,\overline{\overset{\rho}{:} R} \overset{\rho}{\Rightarrow} A[\overline{R}/\overline{Z}]}\text{T-PolyVar}$$

Figure 4.6: Var and PolyVar typing rules.

The terminal rules are T-Var and T-PolyVar, which asserts that a given variable actually exists in the given context. These rules rely on a particular trick inspired by McBride's linear quantitative type system[McB16], which allows a context to be scaled by a factor. The exact usage here of $\rho\Gamma$ means that every element in Γ has usage ρ. This means that for the T-Var rule, the context must consist of the variable in question, and every other value in the context must be unrestricted. This stops linear values being dropped at this terminal proof.

$$\frac{\begin{array}{c}P = \forall\overline{Z}.A\\ \Phi;\Gamma\ [\varnothing]\!\vdash\ n \overset{\pi}{:} P \qquad \Phi;\Gamma', f \overset{\pi}{:} P\ [\Sigma]\!\vdash\ n' \overset{\rho}{:} B\end{array}}{\Phi;\Gamma,\Gamma'\ [\Sigma]\!\vdash\ \text{let}\ f \overset{\pi}{:} P = n\ \text{in}\ n' \overset{\rho}{:} B}\text{T-Let}$$

$$\frac{\begin{array}{c}\overline{\qquad (P_i = \forall\overline{Z}_i.\{C\})_i \qquad}\\ \overline{(\Phi,\overline{Z}_i;\Gamma,\overline{f \overset{\omega}{:} P} \vdash e_i \overset{\omega}{:} C)_i} \qquad \Phi;\Gamma,\overline{f \overset{\omega}{:} P}\ [\Sigma]\!\vdash\ n \overset{\rho}{:} B\end{array}}{\Phi;\omega\Gamma\ [\Sigma]\!\vdash\ \text{letrec}\ \overline{f \overset{\omega}{:} P = e}\ \text{in}\ n \overset{\rho}{:} B}\text{T-LetRec}$$

Figure 4.7: Let and recursive let typing rules.

To bind new values the let or letrec terms can be used. The non-recursive variant, let simply checks that the bound term does indeed match the supplied type, and then proceeds checking the subsequent body of the let term. The recursive variant, which may consist of many sub-clauses, disallows the admittance of any linear values in the given context for the possibly recursive sub-clauses. This stops pre-existing linear values being used in recursive operators and potentially being used many times.

$$\frac{\Phi;\Gamma\ [\Sigma]\!\vdash\ e \overset{\rho}{:} C}{\Phi;\Gamma\ [\Sigma]\!\vdash\ \{e\} \overset{\rho}{:} \{C\}}\text{T-Thunk}$$

$$\frac{\Sigma \vdash \Delta \dashv \Sigma' \qquad \Phi;\Gamma\ [\Sigma']\!\vdash\ n \overset{\rho}{:} A}{\Phi;\Gamma\ [\Sigma]\!\vdash\ \langle\Theta\rangle n \overset{\rho}{:} A}\text{T-Adapt}$$

$$\frac{\Phi;\Gamma\ [\Sigma]\!\vdash\ m \overset{\rho}{:} A}{\Phi;\Gamma[\Sigma]\!\vdash\ \uparrow (m \overset{\rho}{:} A) \overset{\rho}{\Rightarrow} A}\text{T-Ascribe}$$

Figure 4.8: Auxillary typing rules.

The counterpart to T-Switch is the T-Ascribe rule, which is supplied a term and an intended type, checks that the intended type does in fact match the term and outputs the type. T-Thunk transparently unwraps a thunk (a suspended computation) to reveal the naked computation term inside, passing through the type. T-Adapt allows the use of adaptors, of which the full elaboration exists in the original Frank paper. Adaptors allow effects to be duplicated, swapped and hidden, and do not have dramatic ramifications on the linear value system due to commands, the inhabitants of an effect, not requiring usage annotations.

$$\dfrac{\begin{array}{c}(\Phi \vdash r_{i,j} \overset{\rho_j}{:} T_j \dashv\!\![\Sigma]\ \exists\Psi_{i,j}.\Gamma'_{i,j})_{i,j} \\ (\Phi, (\Psi_{i,j})_j; \Gamma, (\Gamma'_{i,j})_j\ [\Sigma]\!\!\vdash n_i \overset{\pi}{:} B)_i \qquad ((r_{i,j})_i \text{ covers } T_j)_j\end{array}}{\Phi; \Gamma \vdash (\overline{r_i} \mapsto n_i)_i \overset{\xi}{:} (\overline{\overset{\rho}{:} T}) \overset{\pi}{:} [\Sigma]B}\,\text{T-Comp}$$

$$\dfrac{}{\Phi \vdash x \overset{\rho}{:} A \dashv x \overset{\rho}{:} A}\,\text{P-Var}$$

$$\dfrac{k(\overline{A}) \in D\,\overline{R} \qquad (\Phi \vdash p_i \overset{\rho_i}{:} A_i \dashv \Gamma)_i}{\Phi \vdash k(\overline{p}) \overset{\rho}{:} D\,\overline{R} \dashv \overline{\Gamma}}\,\text{P-Data}$$

$$\dfrac{\Sigma \vdash \Delta \dashv \Sigma' \qquad \Phi \vdash p \overset{\rho}{:} A \dashv \Gamma}{\Phi \vdash p \overset{\rho}{:} \langle\Delta\rangle A \dashv\!\![\Sigma]\ \Gamma}\,\text{P-Value}$$

$$\dfrac{\Sigma \vdash \Delta \dashv \Sigma'}{\Phi \vdash \langle x\rangle \overset{\rho}{:} \langle\Delta\rangle A \dashv\!\![\Sigma]\ x \overset{\rho}{:} \{\overset{\rho}{:} [\Sigma']A\}}\,\text{P-CatchAll}$$

$$\dfrac{\begin{array}{c}\Sigma \vdash \Delta \dashv \Sigma' \qquad \Delta = \Theta \| \Xi \qquad c : \forall\overline{Z}.((\overline{\overset{\rho_i}{:} A}) \overset{\xi}{:} B) \in \Xi \\ (\Phi, \overline{Z} \vdash p_i \overset{\rho_i}{:} A_i \dashv \Gamma_i)_i\end{array}}{\Phi \vdash \langle c\,\overline{p} \to z\rangle \overset{\pi}{:} \langle\Delta\rangle B' \dashv\!\![\Sigma]\ \exists\overline{Z}.\overline{\Gamma}, z \overset{\pi}{:} \{(\langle\iota|\iota\rangle \overset{\xi}{:} B) \overset{\pi}{:} [\Sigma']B'\}}\,\text{P-Command}$$

Figure 4.9: Command and pattern rules.

Finally there is the T-COMP rule, which describes the relationship between a computation's arguments and its body. This rule relies on a number of pattern rules which produce a modified kind environment and variable context.

These rules have been implemented on top of the existing Frank reference implementation `https://github.com/treloar/frank`.The implementation was straightforward, and the rules as is did not need to be modified.

Chapter 5

Related and future work

The fundamental problem of how to safely deal with resources, such as file handles, that can end up being spoiled has been approached in the algebraic effect language community from a number of directions. Two such efforts have been influential in our choice of resorting to linear types to attempt to provide a solution for this issue. Both of these approaches curtail the use of the continuation variable, which alleviates the need to render the continuation variable linear in relevant situations.

5.1 Runners.

The first solution intended for this thesis to deal with the issue of how to safely handle these perishable resources was to incorporate the runner construct forwarded by Ahman & Bauer[AB19]. On the more theoretical side of algebraic effects, the underlying categorical construct is the model[PP03], which provides the necessary formalisations of what it means to combine and use algebraic effects. Runners are based on the comodel construct, and are used to model resourceful effects.

To define a runner that handles effects, instead of interpreting operations, a runner will consist of a kernel that interprets stateful co-operations. These stateful co-operations map an operation to a single value, and also have the ability to retrieve and manipulate the state of the runner. This eliminates the possibility of either non-determinism or exception handling using a runner, as there is no mechanism to resume many times, or to cease computation. The commands of an effect return a value and internally perform some transformation on the implicit state.

To cover the issue of exceptions being thrown by the linear resource, runners also provide a robust mechanism for handling errors that occur in otherwise linear computations. This effectively hardwires the exception computational effect into the semantics of the language introduced, and covers the deficit that having effect handlers that can only resume once means that exceptions cannot be implemented within the language itself, as they can in Frank.

The reason why runners were not chosen as the solution to the fragile resource problem in Frank is that it would have represented a regressive programming construct in relation to what Frank already offers, and was deemed to not play to the strengths of the language. Frank excels in streamlining the writing of effect handlers, effect multi-handlers and standard operators into a unified construct. To attempt to overhaul the operator semantics into also being able to act as a stateful kernel would have required a significant change to the way that continuations are provided, and it was discovered that a key prerequisite to crafting runners out of operators would necessitate a mechanism to enforce linear use of the continuation in the event that a runner is being implemented. It would also have necessitated implementing several computational effects at the runtime level, which is not quite in line with the motivations of algebraic effects and effect handlers. This is the original train of thought that brought forward the solution of linear types to constrain the ability of handlers to resume more than once.

5.2 Deep finalization.

Another alternative solution is Leijen's extension of Koka with deep finalization[Lei18]. This work promotes handlers that deal with resources as a first-class value, and like runners allow a handler to contain some parameterized state. In particular, the logic of initialization, usage and finalization are carefully pried apart, such that it can be guaranteed that initialization only occurs once, that usage of the resource can happen an arbitrary amount of times, and that the finalization code is guaranteed to run.

The fundamental addition of the deep finalization approach is to wrap every non-resumptive interpretation of an effect with a call to a similar continuation, a *finalize* continuation. This means that instead of evaluating to value and simply returning it, the parameterized state is thread through a finalization construct, which affords an assured point of execution to close any open resources and handle any errant states.

While there are no static guarantees to enforce it, this semantics of assured finalization can still be somewhat simulated in Frank. This is enabled by the shallow handling of effects, which means that instead a handler continuing with itself as the subsequent handler of effects, a new handler can be used instead. This allows the handler to transition on certain commands to a different handler. For example, with a file I/O effect it could make sense to transition between three handlers, the first representing a pre-open state, the second an open and ready to use state and the last being a post-open state. These transitions are modelled simply below.

```
interface File = open : (String) Unit | write : (String) Unit | close : () Unit

preOpen : { (<File>X) [Abort] X }
preOpen <open  str -> k> = isOpen (k unit)
preOpen <_> = abort!

isOpen : { (<File>X) [Abort] X }
isOpen <write str -> k> = isOpen (k unit)
isOpen <close -> k> = postOpen (k unit)
isOpen <_> = abort!

postOpen : { (<File>X) [Abort] X }
postOpen x = x
postOpen <_> = abort!
```

Figure 5.1: Sample handlers with transitions to model file state.

This handler transitioning logic has the ability to dynamically capture invalid use of the file resource, but is by no means a robust solution. If a separate abort effect is injected using an adapting swap, then it is entirely possible to circumvent the dynamic safety that has been attempted. Alternatively if these handlers are nested inside a handler which is performing non-deterministic choice and running every branch of an ND choice, the state of the handler may (quite correctly) be reset when evaluating each branch, which means closing the file in one branch will not register in the other branches. Again the issue lies with not being able to specify the exact behaviour of how the continuation is used, and how this affects the composition of effects.

Furthermore, while Leijen's deep finalization works well for capturing the case of the finalization of effect handlers, it did not seem clear how the solution could be applied to Frank. The existence of a finalization clause in an operator's definition (the clause where no command is captured but an actual value exists) is already present. Due to the flexibility of Frank operators, linearity was still going to be needed to tame the power of the continuations.

Chapter 6

Analysis of the results.

The question to be asked is: what value does linear types actually bring to the table in the realm of languages that provide algebraic effects and their handlers? The first and somewhat obvious answer is that they provide all the general value that linear types already provide. Linear types in their own right are a valuable construct that allows the programmer to statically ensure that a particular value will either be used once, or not used at all. Disregarding the presence of effect handlers entirely, linear types can be used to ensure that concrete linear values are handled correctly.

This is still a useful proposition to add because it means that instead of needing to hard-code the semantics of resourceful operations such as file handle creation and destruction at a language's runtime level, the language runtime can provide lower level primitives that can be used by the programmer to then implement a safe handler to actually use these effects. This means instead of just providing a file effect that has a handler implemented at the language's runtime level, the underlying primitives `fopen : { (String) 1: FileHandle }` and `fclose : { (1: FileHandle) Unit }` can be provided instead. It does need to be noted that this does not actually provide enough of a guarantee to actually ensure that a file does get closed, as in the presence of effects handlers that do not use their continuation the linear value could be discarded.

The key motivation to bring linear types into Frank was to be able to place constraints on the continuation variable found in command interpretations. Being able to statically constrain the value of the continuation variable to being linear, and then subsequently being able to statically check that it is indeed being exactly once, is a key stepping stone to ensuring that handled commands themselves will continue exactly once.

Without linear types there is no general way that does not rely on specific syntactic constructs to restrain the usage of the continuation in a manner that still affords the general flexibility given by Frank. Potential candidates for this syntactic manipulation could work to a limited extent to provide the same effect of a linear continuation, but a solution based on this approach would end up being highly specific to Frank, and not being generally applicable outside of it's realm.

Linear types themselves have not been applied to the realm of directly algebraically effectful languages. This proof of concept to incorporate linear types into such a language has yielded some interesting observations. The three main points of interest are linear adjustments, linear multi handlers and the indication that linear types in combination with algebraic effects seem to provide a solution that is slightly more general than more specific constructs such as runners or explicit finalization semantics.

Chapter 7

Conclusion

The original goal of this thesis was to introduce runners into the Frank language. Eventually, it was discovered that in an algebraically effectual language with access to a raw resumption variable, it would be difficult and largely non-obvious on how to approach creating handlers with a guarantee that they would run once, and only once. Fortunately, in the quiver of tricks and tools available to programming language type system designer there exists a mechanism for enforcing that a value is only used once. Linear types were identified as a necessary step to take in allowing the expression of these safer effect handlers, in order to build up a foundation where these handlers are not cancelled prematurely, or duplicated arbitrarily.

This thesis has not provided any new proofs or measurements, but has instead focused on a proof of concept in introducing two distinct programming language design ideas in a unique combination that has not been explored before. While the full ramifications of linear types and algebraically effectual languages still has a lot of room for experimentation, the hope is that this thesis will provide at the very least an initial insight into the possibilities available when these two notions are combined.

Bibliography

[AB19] Danel Ahman and Andrej Bauer. Runners in action, 2019.

[AHC$^+$16] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. *ACM SIGARCH Computer Architecture News*, 44(2):175–188, 2016.

[BP15] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, Jan 2015.

[DK19] Joshua Dunfield and Neel Krishnaswami. Bidirectional typing. *arXiv preprint arXiv:1908.05839*, 2019.

[Hil] Daniel Hillerström. *Handlers for algebraic effects in Links*. PhD thesis.

[HL18] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *APLAS*, 2018.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[KLO13] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *ACM SIGPLAN Notices*, 48(9):145–158, 2013.

[Lei14] Daan Leijen. Koka: Programming with row polymorphic effect types. *Electronic Proceedings in Theoretical Computer Science*, 153:100–126, Jun 2014.

[Lei18] Daan Leijen. Algebraic effect handlers with resources and deep finalization. Technical Report MSR-TR-2018-10, April 2018.

[McB16] Conor McBride. I got plenty o'nuttin'. In *A List of Successes That Can Change the World*, pages 207–233. Springer, 2016.

[Mor16] J Garrett Morris. The best of both worlds: linear functional programming without compromise. *ACM SIGPLAN Notices*, 51(9):448–461, 2016.

[OLEI19] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, 2019.

[onc]

[PP03] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:69–94, 02 2003.

[PP13] Gordon Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), Dec 2013.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. Citeseer, 1990.