Comprehending Pure Functional Effect Systems

Daniel Tattan-Birch

Supervised by Jeremy Gibbons

Kellogg College
University of Oxford

A dissertation submitted for the
MSc in Software Engineering

Abstract

In order to solve real world problems, software engineers need the ability to represent and compose computational effects such as state, errors and I/O. Functional programming languages support this endeavour through expressive *effect systems*. In particular, modern *pure* functional programmers are afforded a comprehensive toolkit for building applications, including: monadic effects, monad transformers, tagless final, free monads and, most recently, *extensible effects*.

This abundance of techniques is not without cost: comprehending the relative merits of these frameworks can seem daunting. Many of the concepts overlap and the suitability of a specific tool can vary based on the circumstances. Consequently, we may be sceptical of new approaches once familiar with a particular programming pattern. This is epitomised by the continuing prevalence of monadic effects, monad transformers and *MTL-style* programming. Though these have proven to be highly useful techniques, they are not connected *intrinsically* to effectful pure functional programming.

We perform a holistic assessment of established pure functional effect systems from first principles in an attempt to illuminate this territory. Our approach consists of a comparison by example, accompanied by an account of the emergence of effect systems through time; from academic origins to modern software engineering best practices. A simple application, incorporating key practical classes of effects, is implemented using the following effect systems: a monolithic effect monad, monad transformers, free monads and extensible effects.

It is concluded that extensible effects broadly subsume its predecessors, overcoming their limitations and inducing a compelling programming pattern for general application development. There is less demand for MTL *or* extensible effects when writing pure functional programs in a hybrid language such as Scala, due primarily to alternative means of dependency injection. Regardless, transformers and free monads remain useful tools in certain situations.

Contents

1 Introduction

1.1 Motivation

Throughout its rich history as a research domain, the evolution of effect systems has been highly influential on the design of functional programming languages. Regardless of the programming paradigm, there is often tension between academia and industrial software development which stems from fundamentally different incentives. Academics tend to strive for the panacea; general models which satisfy mathematical properties, enabling formal reasoning. Specific effects and programming language constructs can then be specialisations of these models. Conversely, industry programmers care whether the tools at their disposal empower them to solve real world problems in a timely and maintainable manner.

The process of iterating on academic innovations until they generate usable programming patterns has been particularly prevalent in pure functional programming. The implications of referential transparency with respect to effect tracking have given rise to a symbiotic relationship between the academic and programming communities. The design space for effect systems in Haskell exemplifies this collaboration; gradually converging on yet more expressive and ergonomic ways to represent and compose effects.

However, even in this context there can be scepticism towards new approaches. *MTL-style* programming is tried and tested in industry Haskell. This is a framework in which effect operations are encapsulated by type class abstractions and interpreted using *monad transformers*. When combined with pragmatic design choices, this induces a compelling effect system. Once familiar with the constituent concepts and their application to common problems, we may become calcified in our methods and resistant to change. Academic arguments for new programming techniques do not always resonate for engineers since the motivations may not be considered practically significant. As the saying goes, "if it ain't broke, don't fix it". Therefore, we must evaluate these tools through a practical lens as a supplement to the academic foundations.

Extensible effect systems have recently gained adoption in production Haskell systems and are claimed to have several advantages over traditional approaches. In the last few years, a progression of libraries based on extensible effects have been developed, each improving upon the previous in some dimension. There has not yet been an assessment of these frameworks in the literature based on a larger example, more closely resembling real world software systems. This caps off a comprehensive toolkit afforded to modern functional programmers for solving real world problems, including: monadic effects, monad transformers, method dictionaries, tagless final, free monads, coproducts, freer monads and extensible effects. These tools have given rise to different patterns for building applications in languages such as Haskell, Scala and OCaml.

Having such a plethora of techniques is not without cost: comprehending the relative merits of these various effect systems and programming patterns can seem daunting. Many of these concepts overlap and are suited to different situations. For instance, functional programming and monads are now often uttered in the same breath. However, paraphrasing Wadler who married the two, monads are *helpful but not necessary* for pure functional programming [1]. A similar sentiment was echoed by Liang et al. and Jones in their original formulation of monad transformers for *composing* monadic effects [2, 3].

1.2 Contributions

We attempt to provide clarification by giving a holistic assessment of the most prevalent pure functional effect systems. A guiding tenet of this project is to try to understand the current state of the field by examining the context in which discoveries were made throughout its history. Although hindsight bias can cloud it, science tends to progress through a series of breakthroughs building on top of one another. The shape of a domain can be a fairly incidental artefact of the "random" trajectory it took through time: an example of the Matthew effect of accumulated advantage [4]. In particular, the emergence of specific programming patterns can be viewed as a product of earlier innovation.

With this in mind, our approach can be summarised as a whistle-stop tour of the progression of effect systems through time; from their academic origins to modern software engineering best practices. The classical problems of representing and composing effects are examined from first principles, deriving criteria upon which to base the analyses. A simple application is then implemented in each effect system, incorporating four key classes of effects: environment, state, errors and I/O. This is accompanied by commentary contrasting the approaches with respect to the underlying criteria, from the perspective of a working functional programmer.

Given its key role in the evolution of pure functional programming, we primarily use Haskell as the implementation language. We also discuss the differences in usage patterns when building applications using a pure functional style in a hybrid language such as Scala. Full example code can be found in Appendices A and B.

We enumerate the contributions as follows:

1. An evaluation of established pure functional effect systems from first principles.

2. A more comprehensive example of extensible effects than those presented previously in the literature, with accompanying analysis from the perspective of a working functional programmer.

3. A novel Scala 3 pattern for environment passing, used to aid dependency injection.

1.3 Structure

Our comparison by example is performed broadly in line with the chronology of these effect systems. Each built upon its predecessors and influenced subsequent developments. The structure of this document is summarised as follows.

Chapter 2: Background. In §2.1 we begin by defining computational effects and effect systems in general. Early research is presented in the context of both imperative and functional programming. We then narrow our focus in §2.2 to pure functional programming and the initial problem of *representing* effects. §2.2.1 enumerates four key classes of effects upon which we base subsequent analyses: environment, state, errors and I/O. Special attention is paid to the foundational problem of reconciling purity with I/O in §2.2.2.

Chapter 3: Representing Effects with Monads. We introduce *monads* as an elegant solution to the representation problem presented in §2.2. In §3.2 we establish the simple example application used throughout, incorporating the effects of §2.2.1. This is implemented using the rudimentary tools of §3.1 to illustrate the mechanics of monadic effects and analyse their limitations: motivating the need for effect *composition*.

Chapter 4: Composing Effects as Monads. We characterise the problem of *composing* effects from first principles by deriving a set of criteria in §4.1. Any candidate effect system must deliver a satisfactory solution with respect to: expressiveness, modularity, extensibility and ergonomics. Early attempts at combining monadic effects, either by *composition* or *transformation*, are outlined and evaluated with respect to these criteria in §4.2.

Chapter 5: Monad Transformers. We present a practical framework based on *monad transformers* as ostensibly the winning solution to the problem of effect composition. This formed the basis of the ubiquitous *MTL-style* programming pattern. §5.2 implements the example of §3.2 a second time, using monad transformers in a manner that exposes its limitations. We establish the pattern for evaluation used throughout the remaining chapters: the benefits and drawbacks of this effect system are assessed against the criteria sets of §2.2.1 and §4.1. Numerous criticisms emerge from this analysis.

Chapter 6: Conventional Wisdom in Pure Functional Programming. Based on conventional wisdom gleaned from the programming community, academic advances and personal experience, improvements are proposed to the original transformer-based implementation in §6.3. Many of the pitfalls are avoided, producing a smaller set of criticisms. Scala snippets are used to illustrate differences when writing pure functional applications in a hybrid language.

Chapter 7: Extensible Effects. In §7.1.1, free monads are presented as an alternative method for writing effectful programs, with an emphasis on *DSL-driven development*. The example is re-implemented in §7.1.2 and contrasted against the two MTL-style attempts.

The least mature category of effect system in pure functional programming, *extensible effects*, is introduced in §7.2. In §7.3 we implement the §3.2 example a final time. We see that many of the problems of earlier approaches are ameliorated by combining some of their best elements; inducing a compelling programming pattern.

Chapter 8: Conclusion. We conclude by providing a software engineer's handbook, outlining the suitability of different tools to different problems. It is asserted that, for general application development in pure functional languages such as Haskell, extensible effects broadly subsume the other approaches by marrying the most useful elements of both MTL and free monadic eDSLs. There is not such a demand for transformers *or* extensible effects in less opinionated languages such as Scala in which we are afforded greater flexibility. They can still be a useful tool in specific situations; particularly in the absence of I/O and for testing.

Future avenues of investigation are suggested in the context of extensible effects. We speculate that algebraic effects and handlers could have similar potential for future adoption in more mainstream programming contexts. They could even act as a conduit to pure functional programming by providing a more accessible framework for explicit effect tracking.

Appendices. In addition to the code of Appendices A and B, we relegate extended explanations to Appendix C. These provide unnecessarily verbose context and are summarised more concisely in the prose. Appendix D contains definitions. Output logs and an API payload from the example program are included in Appendices E and F.

2 Background

2.1 Computational Effects and Effect Systems

2.1.1 Definitions and Intuitions

Computational Effects. It is instructive to build an intuition for what we mean by a computational effect and an effect system in general. A computational effect can be defined as an interaction by a program with elements outside its immediate environment. The term "side effect" is usually used to capture this meaning when calling a procedure or function; something has happened in addition to computing the return value, if one is returned at all. There are some obvious examples, such as network or file I/O. However, side effects are not limited to these; they can be viewed from a particular level of abstraction. For example, updating a CPU cache would not be considered a side effect in a language such as Python, but it might be in Assembly. For memory-managed languages, a garbage collection cycle modifies memory cells in the application—this is, by design, not observable to the programmer. However, mutating a variable within a particular scope is observable and thus a side effect.

Effect Systems. The ways in which programming languages allow programmers to express side effects and manage their execution is an *effect system*. This may or may not involve a static type system and language support to help track effects.

Imperative Effect Systems. Different programming paradigms choose to represent effects in different ways. In most imperative languages there is no distinction between regular *pure* code and certain classes of side effects, such as mutable state or synchronously sending a network request. This is not captured in the function signature; a developer may need to examine the implementation to understand what it does. Convention and documentation can help mitigate opacity. Issues can be further minimised with the careful use of modularity and abstraction to help developers build a mental model of what's being done in different parts of an application. These techniques are core to the craft of software engineering.

Imperative languages do, however, explicitly capture certain effects. They typically provide syntax for expressing effects like error handling and asynchronicity through language level constructs such as $throw/try/catch$ and $async/await$. These result in *coloured* functions [5]. For example, Java checked exceptions are marked in function signatures until they are handled, and JavaScript asynchronicity is propagated up the call stack until awaited. Once committed to a particular "colour" of effect, it spreads until eliminated with, for example, *catch*, or *await*. This duality has been labelled *effect construction* and *deconstruction* in more formal settings [6]. These operations have fixed semantics, though it is not always clear how they compose and this can vary between languages and frameworks.

Reified Structures. A different approach is to *reify* effects as first-class entities in the system. This may be embedded into a language runtime, or provided by a library. For example, Scheme was the first language to embed *continuations* as a first-class object, enabling continuation-passing style (CPS); a mechanism for managing program control flow.

Concurrency and asynchronicity are other important examples of *control effects*. These are typically represented using some notion of a thread as a reified structure, often called fibers. This enables us to have multiple call-stacks whose interactions with each other and the underlying operating system can be managed by the runtime to support these effects. For example, Golang bakes lightweight threads into the language as an effectful primitive. This solves the coloured function problem by being opinionated about the semantics of effects like I/O; essentially eliminating the distinction between synchronous and asynchronous code [5]. Languages like Rust use OS threads directly instead, generating no runtime overhead. Libraries can extend the default language with an asynchronous runtime, using reified green threads [7].

Pure Functional Effect Systems. The focus of this dissertation will be primarily on effect systems in the context of *pure functional programming*. Since purity has profound implications for the representation of effects, it is instructive to first establish a definition. Sabry gave a formal characterisation based on program equivalence under different *evaluation strategies* [8]. However, our treatment of effect systems is focussed on their principles and constructions, whereas evaluation semantics has greater ramifications for their *implementations*. Therefore, although this is an important aspect of language and library development, it is a wide standalone topic which we consider out of scope. Under this assumption, we may discuss languages with varying default and customisable evaluation strategies in the same terms, for instance: Haskell, a *lazy pure* (call-by-need) language, Idris, a *strict pure* (call-by-value) language, and Scala, a *strict impure* (call-by-value) language.

Instead, we use an intuitive but less formal definition of purity which is often given when motivating and teaching functional programming. Pure functional programming can be distinguished from more general functional programming by the property of *referential transparency*. An expression within a program is said to be referentially transparent if it can be substituted for its value without changing the program's behaviour. If this property holds for a whole programming language, perhaps being enforced statically, then expressions, values and programs can be considered equivalent. It is said that we can *reason equationally* since "=" is comparable with the mathematical equals. This

contrasts to the common representation of "=" as the *assignment operator*: overwriting a variable such as a pointer or scalar value.

We now have the sufficient jargon to define *side effect* more precisely: a function has side effects if it violates referential transparency. Based on this notion of equational reasoning, we can think of pure functional programming as *programming with values*. Referential transparency induces benefits for the programmer and the runtime characteristics of programs. We can refactor expressions without fear of influencing program behaviour; this is part of the reason that functional programs tend to be terse. Parallelising a computation typically constitutes simply substituting a function call for its parallelised version. We can memoise function outputs with similar ease. The assumption that expressions cannot contain side effects also gives rise to a number of compiler optimisations such as rewrite rules to in-line expressions. Henceforth, to avoid ambiguity, programming in a referentially transparent manner will be used interchangeably with both *pure functional programming* and *programming with values*.

There are significant consequences of referential transparency for our effect systems. For instance, exceptions as we know them in imperative languages are off the table since they modify control flow in an irreversible way. I/O also presents problems: a function performing I/O cannot be treated as a value. Clearly we need to reconcile side effects with purity.

This inevitably leads us back to reification in a data structure. There are many ways to achieve this, as we shall see in §2.2.2. However, *monadic effect systems* have dominated this space since their inception and inclusion as Haskell's default runtime: the `IO` monad [9, 10]. The Cats Effect Scala library is an example of a non-native reification of a pure functional effect system [11]. Its `IO` data structure uses fibers for cooperative multitasking to share system resources. These are mapped to threads in the underlying runtime, such as the JVM or Node.js.

Algebraic and Extensible Effect Systems. Algebraic effects, handlers, and so-called effect handler oriented programming has risen to prominence in the effect system research domain over the last decade. Originally formulated by Plotkin and Pretnar [6, 12], they are characterised by operations providing *syntax* for use in effectful programs, and effect handlers giving specific *semantics* to these operations. Typically, explicit effect tracking is used for safely raising and eliminating particular operations. Several languages have been developed with first-class support for algebraic effects and handlers [13, 14, 15, 16, 17, 18], in addition to library embeddings for more production-ready languages such as Haskell, Scala and OCaml [19, 20, 21]. The goal has often been to provide an expressive, coherent and safe API for effect handler oriented programming.

Extensible effect systems, introduced by Kiselyov et al. [22, 23], are mechanically similar to algebraic effect systems. Though neither algebraic nor extensible effects are theoretically wedded to a particular programming paradigm, they have generally emerged as alternatives to traditional approaches in different domains. Algebraic effects have been motivated as generalising specific language constructs available in imperative languages. On the other hand, extensible effects have risen to prominence in pure functional programming, gaining adoption in production Haskell systems as an alternative to MTL. Hereafter, we may abbreviate *extensible effect* to *EE*.

With this in mind, EEs fall well within the remit of our analysis of pure functional effect systems. Algebraic effect systems is on its own a wide and nuanced domain. Hence, though there is significant overlap with the topics discussed here, we consider it out of scope. Even so, the criteria established here may be used as the basis of a similar analysis of algebraic effects in practice. There are interesting ramifications of algebraic effect systems with respect to purity; the necessity of referential transparency is arguably diminished with sufficiently powerful effect tracking. Exploring this is planned for future work.

What Constitutes a Compelling Effect System?. In §2.2.1 it is argued that four key effects are the most crucial in software engineering contexts: environment, state, errors and I/O. We have already given a basic intuition about how these effects may be represented in pure functional languages. However, to build real world applications we need to *compose* multiple classes of effects. Chapter 4 examines this problem in depth. Constructing an effect system that facilitates the composition of these effects as well as more nuanced interactions involving continuations and nondeterminism is a difficult problem. The primary goal of thesis is to introduce and evaluate the most ubiquitous pure functional effect systems with respect to these problems of representation and composition. Therefore, a more detailed background for each approach is presented prior to the analyses of Chapters 5, 6, and 7.

To perform our review of functional effect systems, it is instructive to understand the problem space from which monadic effects emerged. We briefly recount the story of early programming languages and the introduction of the notion of an effect system.

2.1.2 Origins of Effect Systems The origins of many modern programming constructs can be traced back to LISP (1958) and ML (1973). They have spawned families containing dozens of languages in the subsequent decades, including many used today. These are both impure functional languages which can be used to produce *mostly* side effect free code by encouraging immutable data, pattern matching, recursion and higher-order functions; while still permitting mutable state and other side effecting I/O when necessary. Nevertheless, they have been highly influential on the evolution of pure functional programming languages in academia and industry.

Effect systems in statically typed, pure functional languages such as Haskell have a rich history as a research domain. In 1989, Hudak gave a comprehensive assessment of the state of early functional programming languages [24]. Although this covered many defining characteristics of functional programming, most still fundamental today, we keep focus on the commentary around effects; preceding the discovery of monadic effects in 1990.

An inflection point for functional programming was John Backus' Turing Award lecture in 1978 in which he strongly advocated for functional programming as a more natural way for humans to conceptualise programs while encoding problem domains for machines [25]. This was in opposition to the design of his own language, Fortran, as well as many contemporary languages such as COBOL, Basic and C. He felt their emphasis on statement-based programming was too coupled to the instruction-oriented environment of the underlying von Neumann computer architecture. The language design space was growing bloated and getting bogged down in details, missing opportunities to reimagine how programs were written more broadly. Given Backus' platform and status, this sparked a flurry of research into functional languages.

By 1987 the disparate areas of research into functional programming had formed a Tower of Babel [24]. This motivated the need for standardised languages that consolidated many of these features, providing a home for research, teaching, and application development. To that end, specifically in the context of *pure* functional programming, Haskell, a lazy, polymorphically typed pure functional language was established in 1988 [26].

The problem of modelling effects in pure languages was an open research area in the 1980s. Gifford and Lucassen were influential in establishing the research problem of statically tracking effects, coining the term "effect system" in 1986 [27]. They sought to reconcile the benefits of pure functional programming with the utility of side effecting imperative languages, in particular with respect to mutable state. *Effect classes* were proposed. These are somewhat analogous to checked exceptions or algebraic effect operations: syntactic markers instructing the compiler that a particular function or procedure may perform a certain type of side effect. Although they focussed specifically on state transformations, the ideas inspired breakthroughs in both functional and imperative contexts [10, 28, 29]. The coloured functions we have today, such as the *async* "colour" first introduced in F# [30], can be traced back to Gifford and Lucassen. *Type and effect systems* were later formalised and generalised [31, 32].

Following this early work, the number of strands of research into effect systems multiplied. This produced innumerable languages and flavours of effect system across different paradigms: static, dynamic, pure, impure, declarative, imperative, object-oriented (OO), academic, industrial and all permutations of such dimensions. Therefore, to maintain focus, we continue the story with pure functional programming in the 1980s.

2.2 Representing Effects in Pure Functional Languages

2.2.1 Criteria for Effect Representation Whereas Gifford and Lucassen were interested in statically tracking effects in impure functional and imperative languages, Wadler, Hudak and others were trying to tackle the thorny problem of achieving this in a pure functional and lazy language [33]. This brought its own unique challenges. Prior to the discovery of monads as a vehicle for effects, there were several proposed approaches which had been explored in languages such as Hope and Miranda [34, 35]—predecessors to Haskell. These were used predominantly for teaching the principles of functional programming, rather than writing "useful" programs; though some were ported to Haskell after its inception [36].

It will be instructive to understand the criteria that must be satisfied by any worthwhile functional effect system. Then we may assess from first principles how these early techniques were improved upon by monad transformers, free monads and extensible effects. We can distil down the practical concerns generated from comparisons with contemporary imperative languages to the following key classes of effects:

A1. **Environment**. Pass shared parameters and capabilities to functions with the ease of imperative techniques such as global variables and function pointers.

A2. **State**. Manage state transformations with comparable ease to imperative mutable state.

A3. **Errors**. Handle exceptional situations without violating referential transparency.

A4. **I/O**. Support I/O operations such as reading from or writing to disks or channels.

Retaining the characteristic compositionality of functional programming was an important bonus. Note that the environment effect, also known as *dynamic binding* or the *reader* effect, is essentially read-only state (a specialisation of **A2**). However, we consider them both fundamental effects and in practice the situations in which they are deployed can vary significantly. *Capabilities* are meant in the sense of modular abstractions, potentially user-defined, that may be passed to effectful programs through *dependency injection*.

We can divorce **A1**, **A2** and **A3** from **A4** because there were already reasonable ways to express these effects. Data was immutable but could be passed to functions, returned from functions, combined and transformed explicitly. Errors

could be represented in types such as `Maybe` or `Either`. These were arguably a little cumbersome to deal with, requiring repetitive pattern matching, folding and excessive parameters in functions; but not a deal-breaker.

I/O was trickier. There was no fully satisfactory technique without either technical drawbacks or reduced expressiveness. A few candidate techniques are presented in the next section.

2.2.2 Early Attempts at Representing I/O We will outline three early solutions to **A4**. They are all equivalent in power and can be expressed in terms of each other, as demonstrated by Hudak [24]. Interestingly, each of these preluded later developments which will be touched upon.

Lazy Streams. The streams model consisted of lazy lists of data objects representing the classes of effects supported by the language [37, 38]. These "events", of type `Response`[1], resembled a *domain-specific language* (DSL) and were modelled using a *sum type*. This is a particular kind of *algebraic data type* (ADT) consisting of a union of data constructors. In this context we can intuit the DSL as a set of instructions for describing effects in our domain: communicating with the OS. We shall see that representing effects as DSLs, specifically DSLs *embedded* in a host language (eDSLs), are intrinsic to free monadic and extensible effect systems, covered in Chapter 7. The key difference here is that they were placed in lists:

```
data Request  = Put c Char | Get c
data Response = OK | OKCh char
type Dialogue = [Response] => [Request]
```

This resulted in some unsatisfactory properties. The order of the list determined the execution of effects when given to the runtime. The positions of events associated requests with responses, which introduced incidental complexity by not accurately enough representing the domain. The programmer must be careful not to force evaluation of a response prior to receiving the corresponding request since this would result in deadlock. Programs built in this manner were also not composable: we cannot combine values of type `Dialogue` [10].

Continuations. Continuations had been a core part of programming language research in general for some time [39] and were now being modelled in pure languages such as Hope [33]. There existed the notion of a continuation-style *transaction*: separate continuations were passed around to be invoked following the success or failure of an effect [34]. Consider this continuation-based file reading representation:

```
ReadFile name (msg      -> error_transaction)
              (contents -> success_transaction)
```

This has better semantics. In fact, in their 1988 assessment of purely functional I/O, Hudak and Sundaresh expressed that the continuation style felt the most intuitive for reading and reasoning about programs [33]. This foreshadowed the monadic programming style, in which we can think of effects as continuations with a *computation wrapper* around the return value. The relationship stands up to scrutiny: Filinski later proved formally that continuations not only form a monad but in fact *generalise* monads; any monadic effect can be implemented in a language containing state and reified continuations [40, 41]. This is manifested as the `Cont` type in the Haskell *mtl* library we deploy in §5.2. Continuation-passing style is a technique still commonly used in both programming language design and practical situations which command the manipulation of control flow, such as coroutines for concurrency.

However, as a general effect system this was too verbose and cumbersome for application development. It required passing around continuations in any effectful computation; antithetical to the conciseness usually associated with functional programming.

Functions. The final approach had been discussed informally for some time, but not explored in more depth before Hudak and Sundaresh [33]. Consider the following effect representation:

```
type IO a = System => (System, a)
```

An effectful computation is simply a function accepting the entire state of the operating system and returning a new system along with a value. Peyton Jones and Wadler later experimented with this formulation in the implementation of Haskell's monadic effect system, characterising the `IO` monad as a function operating on worlds [10]. This had some intuitive semantics but again required manually threading the system state through to subsequent computations.

In addition to the usability issues, these early approaches suffered another core problem: extensibility. Programmers could not easily create their own effects; adding new effectful primitives such as calling OS procedures required changes to the language itself. They did not offer an intuitive and composable method for writing pure programs with I/O, nor did they provide a satisfactory solution for state and error handling. We could do better.

[1] Named because they were, in a sense, *responding* to the operating system.

3 Representing Effects with Monads

Monads are ubiquitous in the modern functional programming vernacular and have been given a new lease of life with implementations in more mainstream languages. However, they are not inherently coupled to either pure or impure functional programming. Rather, monads are a useful tool for sequencing effectful computations; often represented as data structures. The history of programming languages has been littered with serendipitous unions such as the marriage of monads with effectful functional programming. This will become evident as we outline the origin story in §3.1. §3.2 introduces the example used throughout which is then implemented in §3.3 using the first generalised monadic effect system, Haskell's `IO` monad. In §3.4 this is analysed with respect to the four key classes of effects, **A1**–**A4**, and compared against the earlier approaches to I/O.

3.1 Origins

3.1.1 Categorical Origins In the late 1980s, the same time effect systems were being explored within Haskell and other functional languages, Eugenio Moggi was researching computing theory in a mathematical context. The effects we have introduced were referred to as *notions of computation* [42, 43]. The motivation was to develop a formal system for reasoning about both programming languages in general and programs within a fixed programming language. *Denotational semantics* had already been established to provide a rigorous mathematical language to describe programs [44, 45]. However, Moggi felt that focusing on specific types of computations and syntactical noise was obfuscating more abstract structures that underlay programming languages.

The question then became, what kind of construction could describe computations in the most abstract sense? He naturally turned to category theory, which is, in a sense, a theory of theories: "category theory comes, logically, before the λ-calculus" [46]. The main criterion was that this model needed to generalise specific notions of computation, such as exceptions, state and non-determinism, in favour of a more abstract formulation.

Moggi found that monads were a natural fit. He introduced the now familiar intuition that programs are functions from *values* to *computations*, $A \to M\,B$, where A and B are sets of values (types), and M is some *type constructor* (higher-order type) which will vary according to our specific notion of computation [43]. This idea underlying his complex formal exposition was remarkably simple. The immediate application proposed was to use it as the basis of a more modular approach to denotational semantics. Different features of a language could be defined formally in isolation and then composed to build a more complex semantics for whole programming languages.

3.1.2 Monads in Functional Programming Back in the context of functional programming, Wadler sought to find a more satisfactory effect system fulfilling criteria **A1**–**A4**. For pure functional programming to prosper, it was necessary to reconcile effects such as I/O with referential transparency and practical ergonomics. The proverb that *necessity is the mother of invention* was applicable

In the seminal 1990 paper *Comprehending Monads*, Wadler made the breakthrough of applying Moggi's mathematical monadic model of computation to actual functional programs [9]. The connection was made by generalising list comprehensions which already existed in Haskell. Lists continue to be one of the key monads used for building an intuition of these concepts. They also have utility in the interpretation of effects such as *nondeterminism*: we discuss this in §5.3.2.

We briefly introduce the monad as a programming abstraction by giving a minimal definition. We use the modern names, rather than Wadler's original `bindM` and `unitM`. Let M be an arbitrary type which takes a type and produces a type; we call this a *type constructor* or *higher-kinded type*. Monads can be characterised by the following two functions:

```
(>>=)  :: M a -> (a -> M b) -> M b
return :: a -> M a
```

Then M has (or, more often in vernacular, *is*) a monad if it is possible to implement `>>=` and `return` while satisfying the *monadic laws* of identity and associativity. These laws, along with an alternative formulation in terms of `fmap`, `join` and `return`, can be found in Appendix D.1. Common synonyms we will encounter in subsequent sections include `bind` and `flatMap` for `>>=`, `unit` and `pure` for `return`, `map` for `fmap` and `flatten` for `join`.

How do monads address our criteria? We can concretise our abstract M by defining a type alias for each effect. Wadler included implementations of the monad operators, along with their laws. For terseness, type annotations are given in-line in the examples below.

***A1**. Environment & **A2**. State.* Wadler presented a form of the state transformer monad[1]:

```
type M a = State -> (State, a)
```

Simply a function taking an initial state, producing a value and an updated state. We can define operations for manipulating state in terms of this M:

[1] Distinct from the state *monad transformer*, `StateT`, which we will introduce in §5.1.

```
get   = (\s -> (s, s)) :: M s
put x = (\_ -> (x, ())) :: M ()
```

Note that the `get` component of state is sufficient for the environment effect, **A1**. The comprehensive examples of Chapters 5–7 pay this effect more isolated attention.

***A3**. Errors.* Failure can be modelled using `Maybe`, `Either` or a custom sum type, and an operation for raising errors:

```
data Maybe a = Nothing | Just a
type M       = Maybe
raise        = Nothing :: M a
```

The key idea is that the monad implements *short-circuiting* of the computation, as with exceptions in imperative languages. If an error occurs, subsequent effects do not get executed.

We no longer need to thread state variables through functions and can avoid polluting every function with the specific data type vessel used for representing errors. Instead, we program in terms of our monad *M*, along with effect-specific operations such as `get`, `put`, and `raise`. The monad operators enable us to repeatedly chain effects, giving an "imperative feel" [47]. The semantics of `>>=` varies according to the effect: for state, the monadic effect is the carrying of an additional state parameter; for errors, it is short-circuiting. This *M* can later be instantiated to whichever type we like, as long as it supports the operations we have utilised.

What if we want to use state *and* errors in the same program? This is a problem because our *M* is specialised; we need *M* to support both state and errors. We could try passing the state alongside the `Maybe`, updating our functions as below:

```
type M a = Int -> (Int, Maybe a)
get      = (\s -> (s, Just s)) :: M Int
put x    = (\_ -> (x, Just ())) :: M ()
raise    = (\s -> (s, Nothing)) :: M a
```

The structure of *M* guides us toward definitions of `>>=` and `return` to capture this interaction monadically. Will a `raise` following a `put` keep the state, or discard it? The above definition supports the former semantics, which is consistent with the imperative characterisation of exceptions and non-local state. Discarding state is known as a *transactional semantics* and would require a different definition for *M*, with `Maybe` on the outside:

```
type M a = Int -> Maybe (Int, a)
```

MTL and many other modern effect systems offer this by default (to some disapproval [48]). In general, this is the issue of *compositional semantics*. Defining mechanisms that provide satisfactory compositional semantics turns out to be a difficult problem to solve for arbitrary pairs of effects. Chapter 4 will cover this in depth.

For now, assume we have unambiguous compositional semantics. This is still a painful process. What if we want to introduce a third effect? We need to change our effect type *M*, and every operation involving it. The problem is that our effect operations are defined in terms of concrete types. Wadler suggested that perhaps we have a library containing the standard monads and their compositions [1]. However, coupling effects so tightly is nonmodular and introducing custom effects would still be problematic.

3.1.3 The First Monadic Effect System

Solving **A4**, the problem of representing purely functional I/O, consisted of finding a structure to represent arbitrary side effects in a manner comparable to `Maybe` for failure. The monadic technique described by Wadler hinted at how this may be achieved using functions to delay computations [1, 9]. Monadic I/O represented in the familiar form we use today was introduced first by Gordon as part of a prototype applying pure functional programming to the analysis of medical images [49]. At the same time, an efficient concrete implementation was being developed for Haskell at the University of Glasgow in the early 1990s. This culminated in the introduction of the first general purpose effect system supporting monadic I/O: Haskell's `IO` monad [10].

Peyton Jones and Wadler distilled the essence of the problem down to the idea of separating *being* from *doing*. Any sufficiently powerful effect system needs to distinguish between abstractly describing or *being* an effect, and actually executing or *doing* the effect with the semantics of a particular implementation. Purity makes this duality unavoidable, but the idea is intrinsic to effect systems in general and has since been repeated in the context of algebraic effects [6, 50]. This will be reiterated as we move through the different effect frameworks.

Effect monads like `IO` are immutable data structures representing an effectful computation. Building a value of type `IO a`, for some type `a`, does not *do* anything; it's no different to creating a list. To actually perform these effects, we need to *unwrap* this monadic container; *discharging* the effect. Typically, in pure functional effect systems this occurs when we pass the value to the runtime at the *end of the world*: the program's entry point. This means we preserve referential transparency throughout our codebase. With `Maybe`, we can discharge the monadic wrapper using pattern matching or folds. How do we get our value back out of `IO`? Functions are provided to unwrap it and execute the side effects, such as `unsafePerformIO` in Haskell and `unsafeRunSync` in Cats Effect.

Note that the representation of this data structure could vary. Many library implementations such as Cats Effect and ZIO in Scala implement their effect monad as an algebraic data type [51, 52]. Alternatively, it could be modelled as a sequence of functions. In any case, we fundamentally require first-class functions, captured by reified structures called *thunks*, as a means to delay execution. Filinski's proof alluded to in §2.2.2 confirms this intuition: a language providing a place to store data in memory (a storage cell) and reifiable functions (first-class continuations) is enough to implement monadic effects [40].

Where do monads fit into this? If `IO` can implement the monadic operations, then we should be able to sequence them together like the other effects. We have seen that `IO` can be conceptualised as a function operating on worlds: `type IO a = World => (World, a)`. This is structurally identical to the state effect presented by Wadler; we have already established that this as a monad [1]. Intuitively, the `>>=` operation is as simple as carrying the world through function composition, with `return` returning the monadic value and leaving the world unchanged. Hence `IO`, just like state and errors, is another instantiation of our abstract monad *M* supporting chaining of effectful computations.

This is an ideal time to clarify some terminology. The *being*, of Peyton Jones and Wadler [10], is usually represented by syntax for expressing effects within a language; used by the programmer to describe an abstract intention. These are sometimes called denotations or algebraic terms and are used in actual *effectful programs*, such as domain-specific logic in industrial software. Examples include the `get` and `put` operations we have seen. The *doing* is synonymous with constructing an interpreter for our terms, eliminating the effect by providing particular semantics. In a more general sense, these are the *implementation details*. We have already seen some examples of interpreters with the `Maybe` data structure and `unsafePerformIO` function.

In object-oriented languages, the being and doing are separated using modular abstraction. Interfaces define abstract operations, concrete classes give behaviour to those operations. However, since arbitrary effects are permitted, we are limited in the encapsulation and communicative ability of these abstractions. A method that is pure is not distinguishable from one that performs I/O, for example. In the case of asynchronous I/O, we may be forced to deal with a concrete type or the effect's *colour* syntactically; thus leaking implementation details.

We will use these terms throughout our analyses. It is best to demonstrate how these ideas manifest in monadic effects with a practical example.

3.2 Example: Tracking the International Space Station

We will use a simple but non-trivial program to illustrate the essence of different styles of effectful functional programming. The relative merits of these implementations will be contrasted and evaluated as we progress through each effect system broadly in chronological order of their emergence. Haskell will be our primary implementation language because it has appeared consistently throughout the effect system literature and generally epitomises pure functional programming. However, snippets of Scala are used for comparison when differences arise from programming in an impure language.

The specification for the application is to track the speed of the International Space Station (ISS). This is a reasonably instructive example since it is able to incorporate four of the most important effects in real world software applications: environment, state, errors and I/O.

Our approach is to poll a public API on a configured schedule, computing the speed based on its previous position. The HTTP response contains a JSON payload with the latitude, longitude and a timestamp. This can be found in Appendix F. The documentation recommends calling this API at most once every 5 seconds since the Unix timestamp has granularity in seconds [53]. With this, we can compute the rough speed as the distance delta divided by the time delta. The website claims that the ISS moves close to 28,000 km/h; can we verify this?

We define a common module used consistently throughout the different effect systems, containing the data model, speed calculation and utility functions. Here we present only the important elements necessary for following the examples; the full module is included in Appendix A.2.

```
data Config = Config { httpRequest :: Request, totalRequests :: Count, requestDelay :: Seconds }
```

`Config` constitutes a basic *environment*. `Request` represents an HTTP request and is defined by the *http-conduit* library. All other omitted data definitions are simply `newtype` wrappers around primitive Haskell types. Consider the following model for our application's state:

```
data Location        = Location Lat Lon Timestamp deriving Show
data AppData         = AppData { location :: Location, remainingRequests :: Count, speeds :: Speeds }
```

One elegant solution could be to simply generate a list of programs, each fetching the location with delays interleaved between the calls to produce a value of type `[IO Location]`. Then `sequence` this list, returning an `IO [Location]`, and calculate speeds pairwise. However, this does not require state since the programs are independent. To make it more interesting, we attempt to compute the speeds *in one pass*; printing them out as we go. This means each effectful

computation is dependent on the previous for the last known location. With this in mind, we define the function `updateState` below, accepting the new location, old state and producing a new state along with the computed speed for us to log.

```
updateState :: Location -> AppData -> (KmPerHour, AppData)
```

One of the most practically important aspects of an effect system is the expressiveness of function signatures. Here are examples from our first attempt of §3.3:

```
trackSpaceStation :: Logger -> Config -> IO ()
trackSpeed        :: Logger -> Request -> Seconds -> AppData -> IO (Either JSONException Speeds)
```

These signatures will vary according to the mechanics of the particular effect system. In any case, the pattern will always be the same: `main :: IO ()` will call `trackSpaceStation` and *interpret* effects by discharging any effectful boilerplate, such as "running" monads transformers. It can then deliver our final `IO ()` value to the runtime.

Reading and writing effectful, domain-specific programs in a timely manner while minimising incidental complexity is crucial for building maintainable systems. Hence, evaluating the implications of these signatures with respect to expressiveness and usability will be an important part of the analysis.

We present five implementations of this example. These will act as vehicles to both concretely demonstrate each effect system and facilitate the discussion of their relative merits. The sections in which they appear and appendices of the full programs are as follows:

- **§3.3**. Monolithic Effect Monad. Appendix A.3.
- **§5.2**. Monad Transformers, Naively. Appendix A.4.
- **§6.3**. Monad Transformers, Pragmatically. Appendix A.5.
- **§7.1.2**. Free Monads. Appendix A.9.
- **§7.3**. Extensible Effects. Appendix A.11.

Modifications are suggested alongside the prose throughout. §6.3 improves upon §5.2 by making design choices to alleviate many of the drawbacks of transformers. Scala snippets are provided to illustrate potential differences when using an impure, less opinionated language; the full implementation is included in Appendix B.2. Both these examples resemble an attempt at modern best practices. Of course, this is subjective; we discuss the relevant trade-offs for making informed design decisions.

3.3 Example Implementation: Monolithic Effect Monad

The first implementation uses only the `IO` monad and basic functional combinators. We proceed as if we do not know about later techniques for composing monadic effects, such as monad transformers. We do, however, cheat somewhat by using libraries, constructor classes and `do` notation which were certainly not in Haskell in 1994. The following is intentionally not good Haskell; it is first presented objectively before we scrutinise it in the analysis of §3.4.

A key element of the problem domain is the API call, so we begin with a function for fetching the location of the ISS:

```
fetchLocation :: Logger -> Request -> IO (Either JSONException Location)
fetchLocation log request = do
  maybeBody <- httpJSONEither request
  mapM (`tapM` log) (getResponseBody maybeBody)
```

This will be generally the same throughout, though its position within the program will vary. `httpJSONEither` is a library function representing failure with an `Either`, since we may fail to parse the response. We transform the types using `mapM` and `tapM`; combinators for monadic effects. For observability, we also define a `Logger` in terms of the effect monad `IO`:

```
type Logger = forall a. Show a => a -> IO ()
```

This definition says that a logger is a function accepting a type which is printable using the `Show` type class and producing an `IO ()` value. This is used in our effectful functions:

```
main :: IO ()
main = trackSpaceStation print config

trackSpaceStation :: Logger -> Config -> IO ()
trackSpaceStation log (Config request initialCount delay) = do
```

```
  maybeInitialLoc <- fetchLocation log request
  maybeSpeeds     <- fmap join (mapM loop maybeInitialLoc)
  either logFailure logSuccess maybeSpeeds
  where
    loop initialLoc = trackSpeed log request delay (AppData initialLoc (initialCount - 1) [])
    logSuccess finalSpeeds =
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure e = log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: Logger -> Request -> Seconds -> AppData -> IO (Either JSONException Speeds)
trackSpeed log request delay state =
  if remainingRequests state <= 0
  then log "No more requests to send to ISS" $> Right (speeds state)
  else do
    threadDelay $ getMicroseconds $ secsToMicrosecs delay
    maybeLoc <- fetchLocation log request
    either onErrorReturn onSuccessLoop maybeLoc
  where
    onErrorReturn = return . Left
    onSuccessLoop currentLoc = do
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      trackSpeed log request delay newState
```

The first thing to notice is that there is no special interpretation in `main` since `trackSpaceStation` returns a raw `IO` value. The config value (defined in the common module) and logger are threaded explicitly through the program. We can read the logic as follows: first fetch the initial position, feed this to a recursive loop `trackSpeed`, then sleep with `threadDelay` before fetching the next position, computing and logging the speed. This repeats until we hit our request count. At the end we log in the cases of failure or success. Let's see the output:

```
Location (Lat 48.6182) (Lon (-26.3351)) (Timestamp 1664047148)
Location (Lat 48.7286) (Lon (-25.895)) (Timestamp 1664047153)
"Current speed is: 24890.14km/h"
Location (Lat 48.8479) (Lon (-25.4085)) (Timestamp 1664047158)
"Current speed is: 27382.18km/h"
"No more requests to send to ISS"
"Successfully tracked ISS speed, result is: [24890.14km/h,27382.18km/h]"
```

We see the first two calls to fetch the location, followed by speed calculations produced by two iterations of the recursive loop. Happily, the speed is in the right ballpark! Note that the logs for other implementations will be omitted, but are identical modulo the specific data. The output from running the examples together is in Appendix E.1. Though they are hard to discern in this ad-hoc implementation, there are several effects at play which we discuss in the next section.

3.4 Analysis

We examine the effects one at a time with respect to our criteria.

A1. *Environment* Our functions are required to take additional parameters which do not vary throughout the program. The worst offender is `trackSpeed`: the first three parameters are simply threaded through unchanged. As a result, function signatures are verbose and this would become cumbersome as an application grows. This forms our *environment* which we would ideally just have access to *implicitly*. In an imperative setting, variables could be assigned on program start-up, or the configuation could be passed to class constructors and stored locally in objects.

A2. *State* Our state consists of the previous location, the number of remaining requests and the accumulated list of speeds. Transforming these with each recursive call is reasonably terse, but contributes to a verbose type signature. One can see how this could get unwieldy when scaled to a larger application with multiple sources of state. Moreover, what if our situation demands giving multiple threads access to the state? This mechanism does not facilitate concurrent state.

A3. *Errors* We use an `Either` to represent errors, and decide to abort the program if one of the calls fail. The `either` function is used in two places to fold over the `Either` structure to produce a target value; an `IO` in this instance. This is a more concise alternative to pattern matching, but it is still cumbersome to have to do this every time a computation can fail. We are forced to use combinators such as `mapM` and `join` to massage the types. This has better type-safety and is more explicit than throwing an exception, but is arguably less ergonomic. This appears to simply be the cost of referential transparency; however, the example of §5.2 illustrates how usability can be improved by using monad transformers for effect composition.

More pressingly, the types are actually lying to us. In addition to a `JSONException`, the runtime can really throw any exception at us. This can be simulated by removing the network connection: the HTTP call fails and we do not

see the log indicating the error was handled[2]. Appendix E.2 contains the error we *do* see. This is due to the exception being raised *inside the effect monad*: `IO`. Scala suffers from the same problem. More generally, any pure functional program using an effect monad supporting I/O has this second error channel. We could fix this by collecting arbitrary errors into an `Either` and using `join` to monadically combine the two `Either` values (which may have different error types) together, adding more verbosity. §6.3 alleviates this problem by eliminating the second error channel.

***A4**. I/O* There are three I/O effects: logging, asynchronous network calls and semantic blocking. These infect all of our type signatures with the `IO` type constructor, wrapping the return value. However, this is partially just the cost of statically tracking effects. In an imperative language, the side effects a procedure might perform is opaque[3]. Logging in particular is ubiquitous, and it is preferable to avoid passing it explicitly. In practice, this is often considered part of the environment and can be integrated in more elegant ways as we shall see in subsequent examples.

Another issue is the use of the rather crude concrete `IO` monad. This enables functions to perform arbitrary effects, giving them far more power than they necessarily need. We lose the ability to reason about their behaviour; these functions do not communicate much more information than a side-effecting imperative function would. The underlying issue is the lack of modular abstractions to encapsulate these three I/O-based capabilities.

A helpful way of conveying this is by considering how we might test the `trackSpaceStation` function. We are forced to test the guts of `fetchLocation`, rather than having these details abstracted away and passed in using some dependency injection mechanism. This would allow us to define stubbed implementations of these abstractions. Unit testing is impossible without using mocking hacks to modify the behaviour of the underlying HTTP library function. Otherwise, we must provide a real URL, meaning the test becomes more of an integration or acceptance test. This is unnecessarily broad as the system under test; it is preferable to test smaller units of applications in isolation.

A final, more general point is that the way in which these four flavours of effects are represented is sporadic. There is no consistent pattern for expressing effectful computations and therefore no obvious way to extend the program with a new effect. The syntax and semantics are interleaved such that it is difficult to even identify the effects at play. A developer viewing this program for the first time will need to mentally parse the essence of the problem domain from the various implementation details.

This emphasis on coherently expressing the problem domain is epitomised by the notion of a *domain-specific language*. We have already defined the term DSL in §2.2.2, but it will be instructive to also emphasise its practical importance in the context of software development. One can think of a DSL as an *instruction set* for abstractly describing a particular domain. For instance, the state instruction set is characterised by the ability to get and put state. The essence of failure is the ability to raise and handle errors. `Maybe` captures these instructions in a data type; such a construction is often referred to as a DSL's *initial embedding*. When written and interpreted in a host language such as Haskell, we call this an embedded DSL (eDSL) [54].

In modern programming, it is becoming more accepted that the vast majority of software should be written in higher-level DSLs, abstracting implementation details to runtimes and libraries [55, 56]. These contain the mechanisms for adhering to the DSL description, such as lower-level concerns like concurrency and asynchronicity, modes of communication and specific technologies. This idea echoes Backus' much earlier sentiment about programming conceptually for humans, not for computers [57]. In a pure functional language, the eventual *target* for any DSL is some `IO`-like effect type. This preference for *DSL-driven development* will be reiterated throughout the appraisal of different effect systems.

As we have alluded to, this particular implementation is notably lacking any semblance of a DSL. Clearly DSL-driven development is off the table with this style of programming.

That was a reasonably damning assessment; what are the positives? We have achieved our goal of writing purely functional I/O with an imperative feel. Using `do` notation allows programs to flow in a familiar way and this is certainly more expressive than the approaches involving streams, continuations and functions.

The notion of *temporality* is a core tenet of effectful programming. The behaviour of a program is intrinsically linked to the order of effects within it: changing the order changes the program. Whereas in imperative programming the sequencing of effects is done implicitly through statements, traditionally using the semicolon as the *sequencing* operator, the `do` and `<-` syntax are able to embody this in a pure functional setting [24]. This is why monads have often been described as the "programmable semicolon" [58]; the semantics of sequencing can be modified based on the particular monadic effect.

The significance of the order of sequencing was emphasised by Peyton Jones and Wadler who stated that, in practice, we may interpret an effect monad like `IO` and still maintain referential transparency [10]. The important property to uphold is that executing apparent side effects has no impact on the *behaviour* of the program, so that equational reasoning still applies. One such situation in which we may feel comfortable unwrapping an `IO` is an interoperating

[2]This library provides both simple and more comprehensive HTTP clients: we use the simple one for illustration. As software engineers we must generally assume the worst from internal and external code APIs.

[3]Java's `URL` class notoriously did a DNS lookup in its constructor, meaning the `equals` method can fail if the host isn't resolvable.

call to an obviously pure C function, such as an arithmetic calculation used for efficiency reasons. Having said that, this is rarely necessary in application development, and it is easier to reason about programs that let the runtime interpret this data structure.

Other key advantages of representing effects with monads:

- Monads are composable, which has always been a core principle of functional programming. We can build larger effectful programs, any value of type `IO`, from smaller ones. Note we are talking about value-level composition rather than composition of different *classes* of effects.
- They are extensible in two ways. The `IO` monad can be used as a foreign function interface to interoperate with impure code in lower level languages like C, without needing to change the language itself. Secondly, programmers can create custom monads with their own operations. Of course, this had limited practical utility without a mechanism for *composing* different monadic effects.
- Representing computations in the `IO` data structure had a lower performance impact than might be expected, thanks to compiler optimisations such as in-lining at call-sites [10]. Here we are starting to see some practical benefits of referential transparency.

We have alluded to the necessity of composing different classes of effects such as the four from §2.2.1. Many of the problems presented in the context of the example stem from the lack of a general mechanism for composing monadic effects. Indeed, this was acknowledged following the discovery of monads in functional programming; concrete combined interpreters were unsatisfactory because they lacked extensibility. We address the problem of effect composition and outline some proposed solutions next in Chapter 4.

There is no doubt monadic effects were a major breakthrough in making the leap from pure functional programming being a toy to a valid alternative to imperative programming. However, it's worth reiterating Wadler's view, a sentiment shared by Peyton Jones in this early work, that monads are *helpful, but not necessary* [1, 10]. This has perhaps been taken for granted in recent years: pure functional programming and monadic effects are often defined synonymously when they are not intrinsically linked. Rather, monads are a sequencing tool which have proved to be a highly effective foundation upon which to solve many classes of practical problems elegantly. We keep this in mind as we continue to traverse the literature, maintaining focus on the underlying goals.

4 Composing Effects as Monads

In order to have practical utility for building software applications, we need to be able to *compose* several classes of effects. At least: environment passing, state transformations, errors and I/O. Ideally we should retain value-level compositionality, provide a means of modular abstraction, and build a more consistent framework rather than haphazard patterns for combining specific classes of effects. Given only the tools we have from Chapter 3, an interpreter of effectful programs must be represented as a single monolithic data type. If we want to compose different monads, we must decide how each effectful operation interacts with other effectful operations and manually update both the interpreter and the DSL operations. This is nonmodular since effects are tightly coupled. We illustrated this problem with `get`, `put` and `raise`.

This is not a new problem; it was first observed by Reynolds in 1975 [59]. The most appropriate characterisation in our context was outlined by Wadler in 1998: he labelled it *the expression problem* [60]. There are two dimensions: operations in our DSL are "rows" and different interpretations of these operations are "columns". We would like to both extend the DSL and add new interpretations without needing to modify existing ones. That is, our solution must be *modular* and *extensible*. The relevance of the expression problem will become apparent as we see more examples violating its premise. Therefore, we keep this principle in mind throughout our analyses. At present, clearly our current toolkit is not sufficient. Again, we could do better.

4.1 Criteria for Effect Composition

Similar to §2.2.2 where early approaches to pure functional I/O were outlined, we proceed by traversing the design space for effect composition which culminated in the technique of monad transformers. As before, we derive criteria for a theoretical solution by drawing upon common themes across the literature as well as personal software engineering experience.

B1 **Expressiveness**. Allow programmers to express all the important classes of effects and their interactions with maximal flexibility.

B2 **Modularity**. Effects should be *encapsulated*. In particular, adding, modifying or removing effects should not impact effectful programs, operations (syntax) or interpreters (semantics) involving *other* effects.

B3 **Extensibility**. Provide a formulaic process for defining custom capabilities and extending programs with standard effects without impacting existing programs or interpreters.

B4 **Ergonomics**. Holistically, the system should be usable, and, importantly, teachable. In the same way that programming with a single monad enables expressing effects with an imperative feel, allow programmers to *compose* effects with comparable ease to the language-level syntax and semicolon-based sequencing of impure languages.

Software engineering principles such as testability and maintainability are considered to be crosscutting concerns. For example, if we can attain good modularity, we are well on the way to achieving testability. Strong ergonomics and extensibility give rise to improved maintainability. Compositional semantics, introduced in §3.1.2, is regarded as a corollary of expressiveness since it concerns the ways in which we can *express* multiple effects *together*. If an effect system can satisfy **B2** and **B3** then there is a strong argument that it is a viable solution to Wadler's expression problem [60].

It would be neglectful not to mention the topic of runtime performance. We take the view that, given a satisfactory solution which maximises the above criteria, optimisations can subsequently be made to push the boundaries of efficiency. This has been the case in the design space of extensible effect systems: performance improvements such as effect *fusion* and compiler optimisations have come *after* the consensus that this is an effect system *worth* optimising [61, 62]. A more contentious reason for this not being a primary criterion is that a large swathe of modern real world systems—perhaps the majority—are I/O-bound rather than CPU-bound. Therefore, there is a diminished need for achieving performance comparable to low-level languages. The costs of reduced efficiency are dwarfed by those associated with unmaintainable systems. With this in mind, we delay the discussion of performance to the analysis of extensible effects in §7.4.3, having already covered its predecessors.

4.2 Early Attempts at Monadic Composition

The ability to compose monads was discussed at length by Moggi in the same body of work that introduced monads as a way to structure denotational semantics [43]. After all, a language with one feature is not useful: the whole point was to build complex language semantics from constituent language features. Initially, however, the programming research community had been focussing more on monads than the composition problem [2]. Moggi had proposed *monad constructors* as a mechanism for building a composite monad by adding monadic *increments*, each representing a single class of effect, on top of some base monad. There was a theme to the language here: the idea of increments,

layers, blocks or stacks of monads [40, 63]. These continue to be ways of conceptualising monadic composition, but not effect composition in general: algebraic effects, for example, often favour some notion of *sets* or *rows* of effects. This is a symptom of fundamental differences in compositional semantics.

Three early attempts at monad composition will be outlined and evaluated with respect to the criteria. This gives an exposition of some fundamental problems with composing monads. Additionally, it seeks to illuminate how a problem such as this is solved through a non-linear sequence of attempts prior to one deemed satisfactory enough to build upon. In this kind of domain, there is not really a notion of "failure" or "success", only: good enough to be practically useful.

4.2.1 Steele's Pseudomonads One of the first to address this problem in a programming context was Steele in 1993, who sought to *automate* the process of manually modifying operations when plugging in new monads as we described earlier. He introduced the concept of a *pseudomonad* [63]. Pseudomonads are a generalisation of monads where the `>>=` operator defers to an arbitrary underlying monad. Therefore, it can be thought of as having a *hole*, as with Moggi's monad constructors. As Wadler previously parameterised an interpreter by a monad, Steele is parameterising a monad by another monad.

His formulation of monads was a non-standard reification in data types which required existential types not available at the time. In modern Haskell this is now expressible; full definitions are included in Appendix D.3. In summary, a pseudomonad is characterised by functions *pseudounit*, identical to the *return* we have seen, and *pseudobind*, a weaker form of *bind*. A monad gives rise to a pseudomonad if the *unit* and *pseudobind* operators can be defined satisfying the same laws of identity and associativity. Concretely, given data types `Pseudomonad` and `SteeleMonad` reifying, respectively, pseudomonads and monads, we can define a composition operator `&`:

```
(&) :: SteeleMonad q r -> Pseudomonad p q -> SteeleMonad p r
```

If the value returned by this function always satisfies the monad laws, then it is a valid lawful monad. Unfortunately, if the input monad and pseudomonad are lawful, then we can only say the resulting monad obeys the *identity* laws. Associativity must be proven case by case. This is a symptom of the wider problem, proved in a mathematical setting by Beck [64], that monads do not compose *in general*. This was alluded to by several including Moggi [43] and later reiterated by Jones and Duponcheel in the context of programming languages [65].

Now we have all we need to treat monads as building blocks. This was realised in a so-called "tower of types" which stacked effects expressed as pseudomonads, using *pseudounit* to *lift* them up the tower. Functional programmers will be familiar with this vernacular which highlights Steele's influences on subsequent developments.

We review the framework as a whole. Effects are described in a `Routine` sum type with data constructors used to reify each class of effect. A list of these effects are wrapped up to form a `Package`, which embodies one level of the tower. Combinators were provided to create and update packages for constructing subsequent levels. Steele also defined the building blocks for several of the standard effects.

```
complete (nondeterminism (continuations (errors interpreter)))
```

Figure 4.1: Combined Pseudomonad Interpreter

The result is the ostensibly neat one-liner of Figure 4.1, composing effects for nondeterminism, continuations and errors. Note that `interpreter` is the base, or *null*, interpreter and `complete` "tops the tower". Though this formulation did not lead to practical programming patterns, Steele deserves credit because this was ostensibly the first concrete implementation of a composable monadic interpreter.

4.2.2 Jones and Duponcheel's Premonads Jones and Duponcheel approached the problem from a similar angle to Steele, seeking conditions under which monads could be composed [65]. They presented the *premonad*, also known as a pointed functor in category theory [66], which is characterised by the *fmap* and *unit* operations. Like with the pseudomonad, this can be applied on top of another monad to give rise to a composed monad when certain properties hold. Translated from their implementation language Gofer to modern Haskell, we summarise it as follows:

```
class Premonad m where
  fmap :: (a -> b) -> m a -> m b
  unit :: a -> m a
```

This `class` construct is know as a *constructor class* and had been previously introduced by Jones in Gofer [67]. Its purpose is to capture abstractions such as monads in a manner similar to Haskell's type classes; we have seen an example of these in action with `Show` in §3.3. Constructor classes differ in that they accept abstract *type constructors* instead of types. The *m* above can be instantiated to `Maybe`, `IO` or any other type constructor for which the operations can be lawfully implemented. Hence, regular type classes are a specialisation of constructor classes.

Both are a form of *ad-hoc polymorphism*, in essence *operator overloading*: different types can provide different implementations of the same operators. Some prefer this to inheritance-based overloading due to a decoupling of types from behaviours. Enriching a type with new behaviours is noninvasive and we are less likely to create *leaky abstractions*.

This mechanism was later added to Haskell and is still a foundational part of pure functional programming. Other languages use different representations of the same concept, such as Scala traits parameterised over a type constructor. Hereafter we use the terms *type class* and *constructor class* interchangeably since the distinction is generally considered immaterial.

Proving premonad composition was simple: that functors compose, the *fmap*, was well known, and *unit* can be derived from basic equational reasoning. Given this, composing two monads can be *entirely characterised* by being able to implement the following *join* function, satisfying two monadic laws defined in Appendix D.4:

```
join :: m (n (m (n a)))-> m (n a) -- the target function, for monads m and n
```

The process undertaken to achieve this was to "guess" at implementations of *join* via the auxiliary functions *prod*, *dorp* and *swap*:

```
prod :: n (m (n a)) -> m (n a) -- monad m, premonad n
dorp :: m (n (m a)) -> m (n a) -- premonad m, monad n
swap :: n (m a) -> m (n a)     -- prod and dorp can be written in terms of swap
```

Since *prod* and *dorp* can be written in terms of *swap*, the problem is reduced to its essence: if we can implement *swap* for two monads *m* and *n*, adhering to the monadic laws, then *mn* has a monad, and we're done. One might ask then, what is the point of *prod* and *dorp*? These functions, along with their own laws, are essentially weaker conditions under which monads can be composed. Weaker conditions give rise to broader applications. This was demonstrated with the `Maybe` and `Reader` monads.

Each monadic composition consisted of fixing one monad concretely and letting the other vary, again in the spirit of Moggi's type with a hole [43]. Plugging another monad into the hole is like adding a building block to Steele's tower of types [63]. Notably, however, we have improved on Steele's approach by permitting the composition of state (and therefore environment) with arbitrary monads. Additionally, pseudomonads, a *specialised* premonad, were depicted more elegantly using constructor classes along with a composition type class.

We now consider a concrete interpreter written using this approach. Below is a reformulation of the simple *evaluator* implemented by Jones and Duponcheel which incorporates environment, writer—similar to `put` but with in-built state *accumulation*—and errors [65].

```
type Monolithic a         = Reader Env (Writer [String] (Maybe a)) -- monolithic
type WriterMaybe          = SComp (Writer [String]) Maybe          -- first layer
type ReaderWriterMaybe a = DComp (Reader Env) WriterMaybe a        -- second layer
```

Figure 4.2: Premonad Interpreter

`Monolithic` is the monolithic interpreter we have seen before [1]. The type `WriterMaybe` composes the writer and error effects, then `ReaderWriterMaybe` applies the reader effect on top as another layer. `SComp` and `DComp` are data types for *swap* and *dorp* respectively, accepting two type constructors (our constituent monads) and a type (the value inside the monad). Admittedly, this is verbose, but the evaluator presented exemplifies a reasonably digestible toolkit for composition. *right* and *left* functions were defined to embed computations into this combined interpreter; comparable to the now pervasive *lift* characterising monad transformers.

This work was influential in several ways. In §3.4 we highlighted the need for a means of modular abstraction; constructor classes embody such a mechanism. They are now ubiquitous in contemporary functional programming and will appear throughout our effect system implementations. Furthermore, the representation outlined above was used to prove formally that monads do not compose *in general*. Specifically, *join* cannot be implemented for two arbitrary monads *m* and *n* in terms of only the monadic operators without further restrictions. This result, though not novel from a categorical perspective, was important to foster understanding about exactly why composition of monads has fundamental limitations. It has been consistently reiterated within the subsequent literature on effect systems.

4.2.3 Espinosa's Situated and Stratified Monads Espinosa built upon the previous work by emphasising *transformation* rather than *composition* of monads, as Moggi had done in a mathematical setting [42, 68]. This distinction is subtle. The `ReaderWriterMaybe` monad can be viewed as a composition of the monads on the left and the right. The resulting type is a monad only if an instance of the relevant composition type class exists for each of our particular effects. Implementing such an instance necessitates fixing one side of the composition and letting the other be arbitrary. It is therefore not much of a leap to forgo the composition types such as `SComp` altogether. Instead, for each effect, we define a concrete type parameterised over a type constructor and establish that, if the arbitrary type has a `Monad`, then the concrete type has one too. The semantics of this *transformed* monad will vary according to both the specific effect and the semantics of its underlying monad. This is the essence of a *monad transformer*.

Espinosa's system was embodied by a *Semantic Lego* toolkit built from a stack of *situated monads*, forming a *stratified monad* as the final interpreter. Scheme, an untyped LISP dialect, was used to implement the toolkit since treating types as values was not expressible within statically typed languages of the time. We will not attempt to translate this to Haskell.

We can think of situated monads as the building blocks. They combine an underlying effect monad with two pointers, referencing the monads *situated* above and below it in the stack. The stratified monad is then a tower of these blocks enriched with two unique named blocks indicating the top and bottom. This is somewhat analogous to the *head* and *tail* of a doubly linked list. It also feels similar in spirit to Steele's pseudomonads; indeed Espinosa commented that monad transformers are a generalisation of pseudomonads [68].

4.3 Analysis

How do these techniques stack up with respect to our criteria? We assess the relative merits of the three approaches together. We use PS, PR and SS as shorthand to refer to the frameworks involving, respectively, pseudomonads, premonads, and situated and stratified monads.

B1. *Expressiveness* A fundamental limitation of the original pseudomonads was that they could not be used to express building blocks for reading and writing state. The others improved upon this, describing pseudomonads in terms of their own toolkits. Perhaps, therefore, it is a limitation of Steele's particular formulation, not of pseudomonads in general. PS being a special case of a monad transformer, as noted by Espinosa, indicated that transformers could subsume them if formulated in a sufficiently ergonomic way. Espinosa was able to represent continuations but as a *special case* rather than part of the general framework. Although they did not demonstrate it, Jones and Duponcheel's toolkit could be used to model continuations. Therefore, PR imposes the fewest restrictions on expressiveness.

Lists are often used to interpret the *nondeterministic choice* effect. Its composition with other effects was discussed at length in PR and previously by King and Wadler [69]. We outline the problem in greater depth in §5.3.2 when analysing the MTL framework with respect to expressiveness. Suffice to say, lists do not compose with arbitrary monads without further restrictions.

In all three approaches compositional semantics are decided through the ordering of the monads within the composed structure. Specifically, the packages in PS, the ordering in which composition classes were applied to effects in PR, and the inferred ordering of the situated monads in SS. Steele highlights the classic problem that building blocks combined in different orders may have potentially "mysteriously" different behaviour [63]. For example, whether errors preserve or discard state is determined by the order of the `Writer` and `Maybe` effects in the PR interpreter of Figure 4.2. This is acceptable for programming with, provided the framework is digestible enough that the more subtle effect interactions can be understood.

This method of compositional semantics has some negative ramifications. The position of a particular monadic effect within the composition is fixed statically within a program. This inhibits expressiveness by making certain situations impossible to encode; Kiselyov et al. elucidate this in depth [22]. Furthermore, outlining special interaction semantics is unnecessary for certain classes of effects such as *reading* state. Hence, needing to specify ordering merely adds incidental complexity. This is not a problem practically, but perhaps indicates that the approaches we have seen are not as general as they could be. We will see in Chapter 7 that other effect systems can avoid this redundancy.

Finally, we examine the ability to express several instances of the same effect within each system. There is no intrinsic reason to, for example, disallow different types of state or multiple error channels, potentially interleaved with other effects. Steele and Espinosa address this explicitly [63, 68]. In PS, multiple instances of the same effect is not supported; confusingly, the earliest instance of an effect in the list shadows others of the same class. The double environment monad transformer is depicted using the SS toolkit, implying it is possible. However, it is unclear how this would manifest in effectful programs in practice. PR claims that composing a monad with itself can be achieved using the same composition constructions; presumably we would need to define separate operations for lifting effects into the concrete composite monad. For example, in the presence of multiple kinds of state, `get` is ambiguous without the help of some type-level machinery.

B2. *Modularity* In PS, representing the system as a list of effects, along with convenience functions for pulling particular effects out of the list, means that we can modify effects without needing to touch other building blocks. However, this representation also causes a runtime error if one tries to use an effect not defined in the list. This is not a great trade-off.

PR partially satisfies modularity through type class abstractions for composition, meaning the types of effectful programs can be left untouched. It falls down in the effect operations; modifying the interpreter requires refactoring the *lifting* of other effects into the new interpreter by using *left* and *right*. It is noted that an overloading mechanism could be used to remedy this; in essence an abstraction over the effect operations. This foreshadowed the type class abstractions of Liang et al. and Jones [2, 3], used heavily in the example of §5.2 and at the centre of MTL-style programming to this day.

In SS, Espinosa emphasised the separation of the *language constructs* and the monad operators as abstract data types. This echoes the notion of separating the *being* and the *doing* of effects. This division gives rise to some semblance of modularity. However, the fact that each situated monad layer needs to know about its relative position

reveals limitations; effects should ideally not have to care about one another. Modifying one of the layers impacts other layers: this is nonmodular.

B3. *Extensibility* Writing custom effects in any of these frameworks involves creating a monadic effect and trying to compose it with other monads. In PS, to achieve this we need to modify the `Routine` sum type, meaning it must be accessible. The problem is that this is a *closed union*. Open unions form a core part of more recent effect systems such as monad transformers [2], free monads [70] and extensible effects [22]. PR is an improvement: given that we can create instances of the relevant type classes, a new effect can be treated the same as any other monad. Unfortunately, manual lifting is required. In SS custom effects can be defined as a situated monad within a stratified monad, again requiring changes to the interpreter. All of these criticisms reveal that none of the frameworks provide a satisfactory solution to the expression problem: extending the system demands invasive changes.

B4. *Ergonomics* It can be argued that the primary reason for these frameworks not gaining adoption in their early forms was the issue of usability. Although the PS composition one-liner of Figure 4.1 *looks* appealing, the toolkit required to get there was significantly complex, as Espinosa later pointed out [68]. There is a lot of incidental complexity with the list representation causing a lack of static safety, despite being written in Haskell. The use of partial functions make it possible to cause a runtime error by using effect operations but forgetting to add a building block in the correct place. The representation of monads was also unconventional; although PR and SS later gave more familiar constructions of pseudomonads. Given the numerous usability criticisms targetted at monad transformers in the modern literature [22, 71, 72], covered in depth in §5.3, pseudomonads would never have been palatable to a wider audience.

PR could be considered acceptable to program with. However, the framework is verbose. There are several classes to become familiar with and deciding which combinator should be used to incorporate a particular effect in our system may require some thought. As previously noted, this a symptom of using *composition* rather than *transformation*.

SS feels conceptually intuitive, although it is hard to visualise this represented in a strongly typed language like Haskell. The ergonomics leave something to be desired. Although we have presented the stratified monad as a stack or tower, there is no *inherent order* to the blocks, only the names referenced by each situated monad and the special cases of the top and bottom. It would be cumbersome for programmers to have to deal with the "name" pointers in order to create their own situated monad blocks for user-defined effects or make modifications to ordering; though it is suggested that accessing multiple levels through language constructs would be possible.

4.4 Compendium

We summarise the criticisms of these early attempts at monadic effect composition with the following points:

1. Whilst acknowledging its contributions, we can dismiss Steele's pseudomonads (PS) as a practical framework in its initial formulation since key effects cannot be expressed, in addition to questionable ergonomics.

2. Premonads (PR) and stratified monads (SS) enable programmers to express a large variety of effects, although not with perfect flexibility. They are also more verbose than we would like; rife with incidental complexity.

3. Jones and Duponcheel's framework appears to edge the others as the most viable solution to the problem of composing monadic effects. The use of class abstractions is particularly appealing.

It is clear that, while these frameworks represented promising iterations towards a feasible solution, usability remained a challenging problem. In the next chapter we shall see that Jones combined the approaches outlined in PR and SS by contributing towards a more easily digestible effect system based on monad transformers.

5 Monad Transformers

We can now view with fresh eyes the technique which would dominate pure functional programming for over two decades. It is uncontroversial to state that monadic effects and monad transformers have been the most widely used techniques for representing and composing effects for the majority of the years since their inception. Their prevalence can be attributed to satisfying the established criteria sets, **A1**–**A4** and **B1**–**B4** , more convincingly than any other available effect system. As we have seen, the framework familiar to functional programmers nowadays is merely one representation of Moggi's monad transformer abstraction [43]. It just so happens that this particular framework translated to a compelling, comprehensible design pattern for structuring effectful programs.

5.1 Origins

The breakthrough was presented in Liang, Hudak and Jones' *Monad Transformers and Modular Interpreters* in 1995 [2]. This was a terse paper that built upon many elements of the previous foundational work. It was strongly focussed on the practical software engineering component of composing effects, excluding proofs to avoid drawing attention away from the resulting programming techniques. As with premonads in Jones and Duponcheel's *Composing Monads* [65], Gofer was used as the implementation language with heavy reliance on constructor classes as a means of abstraction.

Following in the footsteps of Espinosa [68], transformation was again emphasised over composition. The mechanics, however, differed from the *pointers* situating different effects within a stratified monad. Instead, monad transformers were characterised by the `lift` function:

```
class (Monad m, Monad (t m)) => MonadT t m where
  lift :: m a -> t m a
```

This definition can be summarised as follows. Let `m` and `t` be type constructors. If `m` and `t m` are both monads (defined in §3.1.2), and there exists a function lifting values of type `m` to values of type `t m` satisfying two new *monad transformer laws*, then we say that `t` is a monad transformer. More concisely, `t` is a monad transformer if and only if it has a lawful instance of `MonadT`. By convention, the concrete data type for `t` is named `MyMonadT` for some monad `MyMonad`. Consider the concrete example for the `Maybe` monad:

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

Here `t` is instantiated by the `MaybeT` data type. `MaybeT` is a wrapper around a value with the shape `m (Maybe a)` for some arbitrary type constructor `m`. Assuming the premise that `m` is a monad, it is simple to write a monad instance for `MaybeT` by unwrapping the nested monads, operating on them with `>>=` and `return`, and rewrapping in the transformer. We relegate this instance definition to Appendix D.2. This pattern typifies instance definitions for transformers, with specific details being determined by the interaction of the fixed monad, `Maybe` here, with the arbitrary monad, `m`. Since we know that `Maybe` is a monad from Wadler [9], all that remains is to implement `lift`. In modern Haskell:

```
lift :: Monad m => m a -> MaybeT m a
lift ma = MaybeT $ ma >>= (return . Just)
```

The definitions of `Monad (MaybeT m)` and `lift` adhere to the monad and monad transformer laws respectively, though we omit the proofs here. Let `m` be Haskell's `IO`. From §3.1.3, we know that `IO` has a monad. Plugging this in, `type M = MaybeT IO` is also a monad. Hence, monadic programs can be written directly in terms of `M`; the key difference being that computations in `m` and `Maybe` must be lifted into `M`. Similar definitions were provided for several standard effects: `EnvT`, `StateT` and `ErrorT` for environment, state and errors respectively.

Ostensibly we now have the *modular interpreter* we sought. However, with this formulation, writing programs in terms of a concrete transformer like `MaybeT` has limited extensibility: wrapping our interpreter with another transformer means we must insert `lift` immediately before all existing operations for the program to compile. It would be preferable to write programs in terms of an abstract monad supporting some subset of the effects in our interpreter. This would avoid explicit lifting and increase encapsulation in effectful functions by limiting their power to only the listed capabilities.

To achieve this, one more piece was required: an abstraction for the effect operations. This was solved using type classes to describe *classes* of effects supporting particular operations. Jones elaborated on this as a programming pattern in a follow-up paper [3]. For the above example, we may now write programs abstractly in terms of some monad `m` supporting operations for `Maybe`, `IO`, or both. The result was appealing function signatures such as:

```
effectfulProgram :: (Writer m, Error m, State m Int) => Int -> m ()
```

This was a significant breakthrough in modularity and extensibility. It combines a few concepts. Class abstractions, introduced in Jones' previous work [65, 67], describe the capabilities available to the monad `m`. The mechanism for delivering these effects is *type constraints* on `m`: the comma separated list on the left-hand side of `=>`. The concrete transformer must then provide a type class *instance* with specific semantics. The compiler passes these dependencies *implicitly* through chains of functions, complaining if a constraint is not satisfied. As a result, the effectful program is

decoupled from the interpreter; obviating explicit lifting and allowing the use of operations with standard `do` notation. Therefore, the interpreter can support extra features in addition to those utilised.

At first glance this feels more intuitive than the other approaches we have seen, requiring fewer constituent parts. We assess this claim using the same example of §3.2, combining several effects within a single program. Hereafter, the term *transformer* refers to the specific monad transformer framework of Liang et al. and Jones [2, 3].

5.2 Example Implementation: Monad Transformers, Naively

We present the example written using what is colloquially known as *MTL-style* programming. For reasons elucidated in the analysis, many practitioners deem it bad practice to write programs in terms of concrete transformers. Conversely, some prefer them—accepting the trade-offs and enjoying efficiency gains. Regardless, it is a prerequisite for functional programmers to understand these interpreters by themselves prior to introducing abstractions to aid modularity and extensibility. With this in mind, we begin by illustrating MTL using concrete transformers. The full code is in A.4.

We proceed by giving a broad outline of the implementation, leaving deeper analysis to §5.3. Again we can start by defining a function to encapsulate fetching the location of the ISS:

```
class Monad m => MonadSpaceStation m where
  fetchLocation :: m Location
```

Already this is an improvement: we use a constructor class to abstract the operation, the *syntax* of the effect. Without concern for implementation details, `fetchLocation` can be used in effectful business logic written in terms of an interpreter of the effect. The `SpaceStationT` transformer below embodies such an interpreter:

```
newtype SpaceStationT m a = SpaceStationT {
  runSpaceStationT :: IdentityT m a
} deriving (Functor, Applicative, Monad, MonadReader r, MonadError e,
            MonadLogger, MonadTime, MonadIO, MonadTrans)

instance (MonadReader Config m, MonadError JSONException m, MonadLogger m,
          MonadIO m) => MonadSpaceStation (SpaceStationT m) where
  fetchLocation = SpaceStationT $ do
    maybeResponse <- reader httpRequest >>= httpJSONEither
    (liftEither . getResponseBody) maybeResponse `flatTap` log
```

Already some drawbacks of transformers are apparent; effect interfaces unrelated to this particular capability are present, along with other boilerplate. We address this in §5.3. Defined in A.2, `flatTap` is comparable to `>>=` , but discards the result of the second parameter (`log` here).

Together, the interface and interpreter form a user-defined building block: `MonadSpaceStation` supplies the syntax, the *being* of the effect, and `SpaceStationT` the semantics, the *doing*. The above instance `MonadSpaceStation (SpaceStationT m)` marries the two, providing a concrete instantiation of the class. The implementation is functionally identical to example §3.3; the main difference is the method of dependency injection. How can we insert this block into a larger interpreter? We define an *application interpreter*: a monad transformer stack for all effects in the program.

```
newtype App a = App {
  runApp :: RealTimeT (SpaceStationT (ReaderT Config (LoggingT (ExceptT JSONException IO)))) a
} deriving (Functor, Applicative, Monad, MonadIO, MonadLogger, MonadReader Config,
            MonadError JSONException, MonadSpaceStation, MonadTime)
```

This implementation uses monad transformers for *everything*. This is not typical usage, but serves to demonstrate the essence of the effect system. `App` is our stack, embodying the combined interpreter discussed in the literature. It consists of several layered transformers including the user-defined `SpaceStationT`. The standard environment and failure effects are provided by the *transformers* library via `ReaderT` and `ExceptT` [73]. The classes `MonadLogger`, `MonadTime` and their accompanying concrete transformers, `LoggingT` and `RealTimeT`, encapsulate the other I/O effects: logging and semantic blocking. These are realisations of the *modular interpreters* described by Liang et al. [2] and the building blocks of Steele and Espinosa [63, 68]. We have enough parts to write our effectful business logic:

```
trackSpaceStation :: App ()
trackSpaceStation = catchError program logFailure
  where
    program = do
      initialCount    <- reader totalRequests
      initialLoc      <- fetchLocation
      let initialState = AppData initialLoc (initialCount - 1) []
      finalSpeeds     <- execStateT trackSpeed (AppData initialLoc (initialCount - 1) []) <&> speeds
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure e = log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: StateT AppData App ()
```

```
trackSpeed = do
  state <- get
  if remainingRequests state <= 0
    then log "No more requests to send to ISS"
    else do
      reader requestDelay >>= threadDelay . getMicroseconds . secsToMicrosecs
      currentLoc           <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      put newState
      trackSpeed
```

`trackSpeed` incorporates the final effect from our criteria, state of **A2**. This is represented by `StateT` and gets interpreted in the *middle* of the program with `execStateT`.

A few discrepancies between this and the original framework presented in §5.1 should be clarified. It is not clear how we are able to use effect operations for transformers below the top level and there is no mention of `lift` or `MonadT`. It would be cumbersome to lift every operation since the number of lifts is tied to the position of the transformer within the stack. For the majority of effects, we do not particularly care where exactly they are situated. Even those whose ordering does matter, such as state and errors, we only want to specify the semantics once rather than being forced to lift explicitly throughout. Clearly, we need a mechanism for automating this process.

In fact, the `MonadTrans` derived by `SpaceStationT` is a renaming of `MonadT` also found in *transformers*. Type class abstractions for the standard effects are defined in the *mtl* library: `MonadReader`, `MonadError` and `MonadState` [74]. It is conventional to name an interface `MonadX` for the effect class `X`. To express the invariant that `MonadX` is indeed a monad, we use a type constraint to ensure that `m` may have a `MonadX` instance only if it has a `Monad` instance. Using these interfaces, we can automate derivation for `App` so that we may use effect operations as if we had used the underlying monad directly. The large list following `deriving` automatically generates instances for `App` via the `GeneralisedNewtypeDeriving` language pragma. The particulars of this automation process will be covered in the analysis, since it forms the basis of many criticisms of this transformer-heavy style.

The final piece of the puzzle is how we actually run this program, since the eventual target value has type `IO ()`, not `App ()`. This is illustrated by the `interpret` function:

```
interpret :: App a -> IO ()
interpret prog = runApp prog & runRealTimeT & runSpaceStationT & runIdentityT &
                     (`runReaderT` config) & runStdoutLoggingT & runExceptT & void
```

Figure 5.1: Transformer Final Interpreter

This function performs *effect deconstruction*, tearing down our monad transformer stack by peeling off each layer until we hit `IO` at the base. It is conventional to *run* or *unwrap* a transformer with a function `runX` or `unX` for a transformer `X`. Different implementations of the same interface can be represented by different transformers, such as `RealTimeT` and its complement, `SimulatedTimeT`; useful for mocking out time in tests. Transformers may also be run with different semantics. For example, `MonadLogger` has convenience functions for standard use cases: `runStderrLoggingT` and `runFileLoggingT` operate on a `LoggingT` value to provide alternative implementations in terms of the default semantics. Here we see how this pattern facilitates modular abstraction in a reasonably flexible way.

Jones' paper [3], a complement to *Monad Transformers and Modular Interpreters* [2], is considered the genesis of this pattern and is cited as the inspiration of the *mtl* library.

5.3 Analysis

We reflect upon this implementation with respect to our criteria sets. As with most programming patterns, we are afforded various options which might be appropriate in different situations. We outline these trade-offs and suggest improvements based on our assessment.

5.3.1 Representing Effects

***A1**. Environment* The `ReaderT` transformer and `MonadReader` class form a building block for dynamic binding. It is parameterised over a particular type of environment, here `Config`. Using the `reader` operation and projection functions we can pull pieces from this config. We have achieved the goal of *implicit* environment access. In Scala, we can achieve an identical pattern modulo syntax. However, other mechanisms for environment passing exist in such hybrid languages. This is discussed in more depth in the *best effort* analysis of §6.4.1.

There is one undisputable criticism: we are tied to a concrete environment in the effectful program. This damages modularity since modifying `Config` necessitates changes to functions operating on it. It could be argued, therefore, that for some situations it is not worth adding the `ReaderT` building block just to avoid explicitly threading shared parameters through functions. Functions are well understood while transformers, as we shall see, carry more baggage.

Nevertheless, reader monads generally[1] have intuitive semantics when interacting with other effects; contributing to their usefulness. For example, *dependency injection* is a ubiquitous technique when writing software in any context. This can be regarded as a particular type of environment passing involving the injection of modular abstractions to different components of an application or library. The wide utility of reader monads in supporting this endeavour is epitomised by effect frameworks `RIO` in Haskell and `ZIO` in Scala which explicitly *embed* this effect into the effect monad—inducing a more opinionated effect system than the traditional `IO` monad [75, 52]. Hence, it is broadly accepted that environment monad transformers such as `ReaderT` are a useful tool in a large number of practical situations, as we shall see with *the ReaderT pattern* of §6.2.

***A2**. State* The `trackSpeed` function has a different return type to `trackSpaceStation`, extending `App` with state by wrapping it in `StateT`. Using `get` and `put` is painless and emulates localised mutable state in an imperative language. As a disclaimer: a state transformer is excessive for such a small program and accumulating state using a list and recursion would be more idiomatic. However, it serves to illustrate both the mechanics of state monad transformers specifically and the interpretation of an effect in the middle of a program. We highlight two key causes for concern.

Boilerplate Instance. To obviate explicit lifting of `fetchLocation` into `StateT`, we are forced to define the following instance of `MonadSpaceStation`:

```
instance (MonadSpaceStation m) => MonadSpaceStation (StateT s m) where
  fetchLocation = lift fetchLocation
```

This is perhaps overkill for one call site, but, in general, manual lifting hampers modularity since the number of lifts must change in line with changes to the stacked transformers. Any additional custom effects would necessitate similar instances depending on their position relative to other transformers. Standard operations such as `reader` of `MonadReader` are automatically lifted into `StateT` via instances declared with its definition, using the mechanics described in §5.2. Defining such boilerplate instances is known colloquially as the n^2 *instances problem* and will be reiterated as a pain point throughout this evaluation.

Concurrent State. How would concurrent state work in this framework? Not well: the semantics are confusing and programs produce surprising results [76]. Engineers do not like to be surprised by the behaviour of their programs. The crux of the issue is that concurrency is a specific source of *nondeterminism*, which has non-standard interaction semantics.

Since both of these issues stem from *composition* rather than *representation*, we delay a more rigorous analysis to §5.3.2. Suffice to say, state monad transformers have limitations.

***A3**. Errors* The `MonadError` interface and `ExceptT` transformer supply the syntax and semantics for exceptions. Again `httpJSONEither` returns us a specialised error of type `Either JSONException Location`. Unfortunately, this must now be manually lifted into the abstract monad using `liftEither`. Error propagation is an example of a *control effect*: an effect that modifies control flow. In this case, receiving a `Left` sends control to the nearest enclosing `catchError`. This is achieved by virtue of the derived `MonadError` interface for `App`, inherited from `ExceptT`. The result is comparable to throwing exceptions in imperative languages: this was our objective.

There are modularity gains since the effectful program need not deal with this error: it is encapsulated as an implementation detail behind the interface. That the error was introduced into the system as an `Either` and lifted into `App` are unnecessary details which are abstracted away. Moreover, `trackSpeed` is devoid of any mention of errors since it does not need to take action. It knows only that `fetchLocation` returns an `App Location`. Modularity can be improved further by abstracting `App` away; §5.3.2 makes this adjustment.

This raises an interesting question, when should programs care about errors in the implementations of their constituent effects or capabilities? In imperative languages exceptions thrown by a method may be indicated in documentation, by circumstantial convention[2], or, less frequently, statically in types or language features. In our example, the presence of a monadic vessel indicates the potential for exceptions, while the presence of `ExceptT` in the interpreter signals to the programmer that certain types of exceptions can be raised and caught. This prompts a broader conversation about the most effective ways to model exceptional situations in software.

Consider the example of a web server. When receiving a network request at an *edge* of our application, we likely want to execute some particular business logic in a deeper component. There may be predictable, domain-specific errors that this component is responsible for dealing with. These tend to be called *exceptions* [77]. Unexpected failures, perhaps caused by programmer mistakes, could be labelled *errors*. Note that, since this terminology varies across different technologies, to avoid confusion we will use the terms interchangeably unless indicated otherwise.

We *almost always* want to handle these unknown errors rather than crashing the application. If the program does not handle it explicitly, the web server framework is responsible for converting it to an appropriate error response.

[1] *Control effects* such as continuations can produce surprising results when combined with reader [22].

[2] For example, a function for accessing the first element of a collection inevitably has some notion of failure.

Almost alludes to the important assumption underlying this decision: that the web server will continue to correctly perform its duties following the error. If, for example, the heap of a JVM application grew too large, we would receive a fatal `OutOfMemoryError`. This is out of scope for the web server; handling these classes of errors causes the JVM runtime itself to be unreliable. For example, basic maths operations may behave erratically. Therefore, it is unadvisable to continue serving requests; better to restart the server and diagnose the issue.

This is emblematic of the more general philosophy that components should control precisely the errors that are relevant to their domain. This might manifest in practice as handling *all* exceptions, since, when I/O is involved, we must assume that there are exceptions we cannot foresee. However, if no reasonable action can be taken in an exceptional situation, then we may defer error handling to other components. For more constrained domains, perhaps without I/O, we can be more explicit about the categories of errors. These might be enumerated in a data type to facilitate exhaustive checking by the compiler when handled.

In the context of our higher level domain, we see this kind of division of responsibility. The presence of `catchError` in `trackSpaceStation` indicates that it is responsible for catching exceptions of type `JSONException`, while the implementation of `fetchLocation` is only responsible for *raising* such errors. Therefore, an immediate improvement in modularity would be to split up the operations for throwing and catching exceptions. In our case the action we take upon handling an exception is simply writing to stdout. This can be verified by breaking the configured URL and observing the parsing failure log. We include this in Appendix E.3.

However, we have not solved the more pressing problem alluded to in §3.4: there is more than one error channel. The specificity of `JSONException` gives away this false type safety. We know intuitively that, when I/O is involved, there are many more classes of errors than just parsing. Since we are using pure functional programming, all other exceptions must be embedded into the `IO` data structure. This can be simulated by removing the network connection and observing that the error was propagated up to the entry point, aborting the program without having handled the error. The logs of Appendix E.2 illustrate this.

So, at best we have handled one particular type of error elegantly. Under harsher scrutiny, the use of `ExceptT` is merely obfuscating a wider category of potential errors, giving the programmer a false sense of security. It is certainly in the remit of our effectful program to catch exceptions related to the network connection of the ISS API call.

Adding further convolution, `IO` defines an instance of `MonadError` for a different type of exceptions: `IOException`. In the presence of multiple layers implementing the same interface within a stack of transformers, the deepest instance gets propagated to the top. In this case `IO` would win. Therefore, if we're not careful it is entirely possible to include a completely redundant `ExceptT`. This problem generalises: mistaking which interface instance is being used can lead to subtle bugs [22]. We got lucky here because `JSONException` is a more specific error defined in a separate module.

Although contentious in some contexts, Postel's robustness principle is applicable for these kinds of exceptional situations: we should be liberal in what we accept. It is preferable to assume the worst regarding exceptions produced by client code. The flip side is that we should be as explicit and specific as possible in situations we control and domains we understand well.

Clearly, the error handling in §5.2 needs adjustment. The points outlined above will form the basis of improvements to error handling in subsequent sections. As a rule of thumb: in programs involving I/O, meaning the bottom of the transformer stack is a monad like `IO`, we should pause before reaching for transformers to model errors.

***A4**. I/O* In the implementation of `fetchLocation` we have substituted the concrete `IO` with an abstract monad and the equally crude `MonadIO` class for lifting `IO` operations into the stack. `MonadLogger` and `MonadTime` neatly encapsulate logging and semantic blocking. We have also avoided having to pass through a logger explicitly. This is certainly an improvement on §3.3. Unfortunately, special treatment is needed to generate an instance of `MonadLogger` for `App` to bypass explicit lifting. We use this as an opportunity to explain the cumbersome process for automating derivation, giving rise to the n^2 instances problem already alluded to.

The n^2 Instances Problem. We consider the problem from first principles. In this implementation, there are 5 transformers, associated with 5 classes, plus `MonadIO` and standard categorical abstractions. To give `App` access to all these interfaces, we need a way to generate instances of each class. Note that newtype wrapper is used merely to *automate* these instance definitions; we could use a simple type alias as seen before in Figure 4.2. It is also possible to evade transformers for custom capabilities, so the number of transformers is only a *lower bound* for the number of interfaces.

Then, assuming we want to completely eliminate `lift`, the more general problem statement is: a transformer stack that contains N transformers and needs to support M interfaces must somehow *inherit* at least N instances from the transformers, some constant number of instances from the base monad, and have manual instances declared for the remaining interfaces (at most $M-N$). We assume transformers implement their own interface partner; there are other constant factors like this. Therefore, the number of instances grows in $\theta(N \times M)$, explaining its name.

For the curious, we include a comprehensive explanation by example in Appendix C.1. Here we summarise the issue by examining a specific transformer / interface pair: `RealTimeT` and `MonadTime`. There are two directions.

Firstly, `MonadTime` must be propagated *up* the stack through every transformer *above* `RealTimeT`. This depends on `MonadTrans` being defined for every transformer in our stack. We cannot rely on this because it demands an *overlapping*

instance definition. This is controversial, since type classes should generally be *coherent*: there should only be one valid instance for any type and type class pair[3] We also depend on the module in which `MonadTime` resides, *mock-time*, providing an instance for arbitrary transformers. This is again discouraged, due to inherent limitations of transformers which we expand upon in §6.4.2. In this particular case we get lucky because `RealTimeT` only has one level to jump up to `App`; automation works out of the box.

We are less fortunate in the second direction: the transformer needs to allow every interface *below* it in the stack to pass through it. `RealTimeT` does not derive `MonadTrans`, epitomising why we cannot trust third parties to implement all the required components for MTL to work smoothly. Consequently, we are forced to declare this instance manually, in addition to a `MonadLogger (RealTime m)` instance in order to trigger a *default implementation* on the definition of `MonadLogger`. Both of these are *orphan instances*—they neither reside in the module of the type class nor the concrete type. These are nonmodular and discouraged: changes to external dependencies, such as new instance declarations, can cause compilation failure, or worse, *silent changes to runtime behaviour*.

All of this is inscrutable from a usability perspective and forms the basis of general aversion towards transformer-heavy approaches. It is also unsatisfactory with respect to modularity. Not only does each effect need to know about other effects, which is nonmodular by definition, but automatic derivation cannot always shield this from the user. We require knowledge of the mechanism's inner workings. This includes several strategies by which derivation may be achieved for any given situation and selection of transformers. These strategies often also vary by language [79, 80].

5.3.2 Composing Effects We assess MTL-style programming against the effect composition criteria of §4.1. Transformers inherited problems from the earlier attempts presented in §4.2; further limitations have since been uncovered by both effect systems research and practical application. Using the example as a case study, we discuss these strengths and weaknesses in greater depth. This is not an exhaustive discussion of all aspects of MTL. The most urgent concerns are analysed here and then addressed in Chapter 6 by applying conventional wisdom. This gives room for further scrutiny, unearthing more nuanced issues.

B1. *Expressiveness* All the standard effects can be expressed using transformers: `ReaderT`, `WriterT`, `StateT` and `ExceptT` with `IO` as the base monad. In the absence of I/O the `Identity` monad can be used at the bottom of the stack instead; performing no effects. A marked improvement on the previous approaches is that we can also express the more exotic effects such as continuations and nondeterminism consistently.

Compositional Semantics. As with Espinosa's formulation of §4.2.3, the ordering of transformers determines semantics. The classic example of state and errors can be expressed with nested transformers `StateT s (ExceptT e)` and `ExceptT e (StateT s)` for, respectively, state preserving and transactional semantics. Appendix C.5 contains a program to illustrate this composition for every effect system we cover; this is discussed in §7.4.2. How digestible is this for a user? There is some redundancy by needing to specify the relative positions of effects for which ordering is irrelevant, such as dynamic binding. Generally though, beyond the initial learning phase, it's an acceptable mechanism for expressing compositional semantics.

That this ordering is *static*, however, is an unnecessary restriction. It is surprisingly hard, for instance, to change the semantics of state and errors in the implementation of §5.2. `App`, the carrier of the `MonadError` effect, is composed of transformers, but is itself only a *transformed monad*. There is no hole for a monad to fill. Hence, we would need to make invasive changes to the interpreter. Other examples illustrating the limitations of transformers, with a particular focus on this issue of static ordering, are presented by Kiselyov et al. [22]. These revolve around two of the more challenging classes of effects: nondeterminism and continuations. We discuss these as special cases.

Nondeterminism. Within the effect system literature, *nondeterminism* is a classic example of an effect for which it is challenging to get the desired semantics [2, 22, 23, 56]. It can be characterised as a combination of: *choosing* between two computations, and *failure* [81]. Historically, lists have been used to interpret nondeterminism. Composing lists with arbitrary effects inherited the same problems described in depth in previous work [65, 69]. Consider a naive definition: `newtype ListT m a = ListT (m [a])`. We require that the arbitrary monad `m` is commutative—defined in Appendix D.5—for `ListT m` to form a monad. Otherwise, we violate the associativity law in the composite monad.

In closing remarks, Jones and Duponcheel suggest that functional programmers may be willing to forgo this algebraic property in favour of pragmatic engineering [65]. However, it's worth emphasising that using lists as a model for nondeterminism is an *implementation detail*. There are other models that avoid the restriction to commutative monads [81]. Over the intervening years, several transformers for nondeterminism have been implemented under various aliases [82, 83, 84]; some are reincarnations of `ListT` without its restrictions.

Even with a reliable transformer for nondeterminism, the semantics are challenging. Kiselyov et al. set out to implement resumable exceptions in the presence of nondeterminism [22]. After dodging a few common pitfalls, it is achieved using two error transformers either side of a `ListT`. This reasonably puzzling result epitomises the obstacles one may face when using transformers in more subtle situations. The underlying issue is that the program commanded different layer orders at different points, resulting in the three-layer solution.

[3] Technically non-overlapping instances are a necessary but not sufficient condition for coherence [78].

Continuations. Though they proved difficult for Espinosa's transformers [68], continuations can be embedded in computations using `ContT`, defined in D.6. Practical use cases include resource management, known as *bracketing*, and coroutines for concurrency. In general, situations requiring the manipulation of control flow may entail `ContT`.

Whereas the example using nondeterminism given by Kiselyov et al. can be classified as an ergonomics issue, the second example was flat out unrepresentable using transformers [22]. In order to implement coroutines that require access to an environment, we must compose the reader and continuation transformers. In particular, they must be *interleaved*: two different semantics are required by the same program. The static ordering of transformers renders this impossible to express.

What are the implications of these limitations in expressiveness? In the context of general application development, these effects are rarely called for. Although there are classes of problems for which transformed lists provide the most elegant solution, we can usually make do without explicit nondeterminism. Therefore, avoiding `ListT` with its non-standard semantics is a reasonable recommendation. Similarly, one would seldom encounter situations requiring continuations: these are primarily concerns for library authors and highly specific real world problems. These use cases are of course still important, but less pervasive.

Moreover, it could be argued that situations which *do* necessitate the use of continuations, nondeterminism and other challenging effects are subtle enough to call for serious deliberation and testing regardless of the effect system in which they are represented. Thus, understandably, many do not find arguments based on these limitations compelling enough to drop the tried and tested MTL pattern. They do not constitute a serious mark against transformers *in practice*.

Nevertheless, seeking the panacea is still a noble goal and one that everyone eventually benefits from. It is always preferable to develop general models which can be specialised to more concrete situations. In this context, being able to write mathematical formalisms, software libraries and applications using the same constructions is highly appealing; we shall see that extensible effects support this endeavour. For now, there appear to be more pressing limitations regarding the expressiveness of transformers.

Concurrent State. While explicit nondeterminism can be bypassed in practice, concurrency is a specific source of nondeterminism that is unavoidable in software development. In particular, concurrent state is not easy to model with transformers. In practice, concurrency is supported by the `IO` monad at the base of the transformer stack. Details like coroutines and cooperative multitasking are encapsulated by functions like `forkIO` or interfaces such as Cats Effect's `Concurrent` type class. Therefore, we can reduce this problem to composing concurrency combinators of `IO` with `StateT`. The article that coined *the ReaderT pattern* contains an instructive example which we briefly regurgitate [76].

```
concurrently :: IO a -> IO b -> IO (a, b)
modify :: MonadState s m => (s -> s) -> m ()
```

Assume we have a concurrency primitive `concurrently` which accepts two `IO` -programs as arguments, running them concurrently. `modify` can trivially be implemented in terms of `get` and `put`. Consider the following `IO` value:

```
program = execStateT (concurrently (modify (+ 1)) (modify (+ 2))) 4 :: IO Int
```

The two programs passed to `concurrently` are inferred as having type `StateT Int IO ()`. If this were an imperative language one would expect both state transactions to be applied sequentially, leaving the concurrency mechanism, perhaps a mutex or `volatile` variable, to deal with how exactly that is achieved. Under this assumption the answer should be 7. Unfortunately, as Snoyman explains, it is not. The result may be 4, 5 or 6, depending on implementation details of `concurrently`. Such limited encapsulation and confusing semantics is unacceptable for writing concurrent applications. Whether these are issues of expressiveness or ergonomics is debatable. Regardless, having no reasonable way to express concurrent state is a problem.

Repeating Effects. It is a fair expectation that an effect system permits the use of multiple instances of the same class of effect. For example, global and local state and environment, or custom capabilities parameterised by different types. The technique described in §5.3.1 does not support multiple instances of the same effect interface: each constructor class may only be used once due to *functional dependencies* in their definitions. This makes it impossible to duplicate effects in programs using type constraints; a useful tool for improving modularity exemplified by the `MonadSpaceStation` instance. It is only expressible in MTL using concrete transformers and explicit lifting, or proxy classes. In practice, it is usually possible to achieve our goals without needing this feature. Nevertheless, it is an unnecessary restriction and merely a product of language machinery.

***B2**. Modularity* Transformers were motivated in the early literature as a tool for *modular interpretation*, a means to describe and compose building blocks for different monadic effects [2, 68]. There is certainly a clear pattern for constructing such building blocks, and indeed for composing them. Unfortunately, we have seen in §5.3.1 that programs written using transformers are not so modular. Explicit lifting is inherently anti-modular, since reordering the stack might demand, for instance, changing a `(lift . lift)` to a `lift` to satisfy the compiler. This problem grows as a function of the number of interpreters and is compounded by the inclusion of custom effects. Hence, we create interfaces and use automatic derivation to alleviate this burden, giving rise to the n^2 instances problem.

Using concrete transformers with automatic lifting still has disadvantages. We have more power than we need in different parts of an application. The `trackSpeed` function can perform arbitrary effects through the `liftIO` operation of `MonadIO`. Readability is damaged: this type signature contributes little to our understanding of the program's behaviour. We are reversing some benefits of the modular class abstractions introduced by Jones [3].

In the spirit of DSL-driven development, it is preferable to structure applications with core business logic in the centre, implementing higher-level custom capabilities in terms of lower-level ones, until eventually we hit the most primitive effects. For pure functional programs, the lowest level is `IO`: a foreign function interface (FFI) with no limits in power. It is desirable to reflect this structure explicitly and statically in the effect system.

This turns out to be surprisingly simple to remedy. We can regain the benefits of modular abstraction by using an abstract monad and type class constraints. Transforming a program to this end is typically formulaic: replace the transformer in the return type with an abstract type constructor `m` and feed in the constraints until the compiler is happy. We can reduce repetition and verbosity of type signatures by grouping common dependencies with type aliases:

```
type MonadAppCommon m = (MonadSpaceStation m, MonadReader Config m, MonadLogger m, MonadTime m)
```

This particular grouping serves only to illustrate the idea; in general one could choose more appropriate groupings. For example, monitoring, used for observability in production systems, could be captured by a `MonadMonitoring` constraint; grouping effects such as logging, metrics and alerting. Similar techniques exist in Scala as we shall see in §6.4.1 and Appendix B.3.1. Using this grouping, consider this modification of the type signatures:

```
trackSpaceStation' :: (MonadAppCommon m, MonadError JSONException m) => m ()
trackSpeed'        :: (MonadAppCommon m, MonadState AppData m) => m ()
```

Remarkably, no changes are needed to the function bodies. The compiler infers that all the operations originate from the same monad. Appendix A.4 contains the full refactoring alongside the original.

We now have some options with respect to the state effect introduced mid-program. `StateT` may be included explicitly in the return type of `trackSpeed'`, or be inferred as the supplier of a `MonadState` instance through the use of `execStateT`. We also have a familiar choice regarding how to propagate `fetchLocation` into this transformed monad: either through explicit lifting or by defining an instance of `MonadSpaceStation (StateT s m)`. Our decision is reflective of the extent to which MTL solves Wadler's expression problem [60]. Using a concrete transformer and explicit lifting directly violates extensibility. Automating lifting is more indicative of a lack of modularity; the instance is an extraneous detail and purely an artefact of the mechanics of MTL. In theory, we would like to use execStateT and have it "just work". Extensible effects achieve this without the incidental complexity. Using an abstract monad is arguably the best option: from the perspective of `trackSpeed'` we can use the state operations by adding only a type class constraint.

We have gained the additional advantaged of highlighting any constraints which were not used, meaning a function had more power than it needed. For example, no exception management was needed in `trackSpeed` and this is now reflected in the constraints of `trackSpeed'`. This is a natural extension of the principle of least privilege: a component of a system should have precisely the amount of power necessary to carry out its duties. The ability to mentally parse functions by looking at constraints pays dividends across larger codebases. This is a common method of abstraction in pure functional Scala and is closely related to the *tagless final* pattern for writing eDSLs; covered in §6.1. Suffice to say, with this technique it is possible to avoid transformers altogether—reaping the rewards of modular abstraction without the disadvantages of interpreters represented as transformers.

However, constructor class constraints are not a panacea. By virtue of referential transparency, Haskell's compiler (GHC) is afforded numerous opportunities to improve performance by rewriting programs. However, the mechanics of type class passing often violates the conditions under which these optimisations can fire [85]. Concrete transformers are far more likely to benefit from these performance gains. We elaborate on efficiency concerns in §7.4.3.

Based on the particular circumstances and these modularity trade-offs, the programmer can choose a flavour of MTL: concrete transformers, with or without a `newtype` wrapper, class constraints over an abstract monad, or some combination of these within the same program. In any case, the principle of modularity is always violated to some degree in the interpreter: changing an effect requires changes to elements of the effect system involving other effects. This is nonmodular by definition. To at least achieve modularity in the effectful program, we would generally opt for an abstract monad since we pay the price of n^2 instances regardless.

***B3**. Extensibility* The `MonadSpaceStation` interface and `SpaceStationT` transformer constitute a building block for our user-defined capability. We can derive a recipe for generalising this process.

The Monad Transformer Extension Recipe. We have succeeded in describing a formulaic process for defining new effects and integrating them into an existing system:

1. Write a class interface: `MonadSpaceStation`
2. Write a transformer interpreter: `SpaceStationT`

3. Write an implementation of the interface for the transformer:
   ```
   MonadSpaceStation (SpaceStationT m)
   ```
4. Define instances of the new interface for the existing transformers:
   ```
   MonadSpaceStation (RealTimeT m)
   ```
5. Define instances of the existing interfaces for the new transformer. This list:
   ```
   deriving (Functor, Applicative, Monad, MonadIO, MonadLogger, MonadReader Config,
       MonadError JSONException, MonadSpaceStation, MonadTime)
   ```
6. Insert the transformer into the stack such that its dependencies are satisfied:
   ```
   RealTimeT (SpaceStationT (ReaderT Config (LoggingT (ExceptT JSONException IO)))) a
   ```

Although formulaic, this is no small undertaking. Introducing a standard effect into an existing system is less cumbersome since a lot of the work is done for us. Depending on the other transformers in the stack, only 4, 5, and 6 may be required. Thererfore, in practice this boilerplate scales with the number of *custom* capabilities.

Dependency Graph . Standard transformers are generally wrappers around a particular monadic effect inside an arbitrary monad, to produce a transformed monad. Their semantics are captured in their structure. Custom effects are slightly different. The instance of `MonadSpaceStation` uses other operations such as `reader` as part of the implementation of `fetchLocation`. This is an example of using a standard effect to implement a custom one, in a sense trading lower-level effects for higher-level, more *domain-specific* effects. Effects such as these are often called *capabilities*, with *capability passing* being another name for *dependency injection* often used in an OOP context. Viewed holistically, this gives rise to a *dependency graph* to capture the relationships between different components of an application. In OOP, tools such as UML diagrams are used to visualise this concretely; more generally it is a mental model used by programmers when developing a system.

Regardless of the programming paradigm, name, or specific mechanism, dependency injection is essentially our main tool for reducing the complexity of software systems: create an abstraction which encapsulates some behaviour, pass this as a dependency into other components, repeat until we have the pieces to glue our system together. This applies to both software libraries and applications, only that libraries usually begin at a lower level. Each component can be unit tested in isolation, with integration tests for larger subsets of the application.

In MTL, the dependency graph can be inferred from the transformer stack. For example, to derive an instance of `MonadSpaceStation` for `SpaceStationT`, we require that the constraints are satisfied. The order of the transformers embodies these relationships: `ReaderT`, `ExceptT`, `LoggingT` and `IO` must all be deeper in the stack than `SpaceStationT`. This method suffers from redundancy and is arguably indirect. The order of those dependencies among themselves is irrelevant; as is the position of `RealTimeT`, currently the outermost transformer. At first glance it is not clear which parts of the stack have intentional ordering for semantic reasons and which are merely incidental.

Moreover, there should be no other restrictions on the position of `SpaceStationT`. The two non-orphan boilerplate instances of `MonadSpaceStation` for `RealTimeT` and `StateT` are necessary only for *this particular ordering*. In general, custom effects should pass through *any* transformers: we're back at the n^2 instances problem.

Overall, we have a formula, but it leads to serious usability issues which we now discuss in isolation.

B4. *Ergonomics* Our final criterion considers the overall usability of MTL as a programming pattern. Starting with the positives: the system as a whole is broadly formulaic, with well established techniques for effect composition and conventions to alleviate some of the burden on the programmer. To their credit, transformers are also not a huge conceptual leap from monads. This was an outright improvement on premonads [65], pushing the lifting of operations to the interpretation through class abstractions. As pointed out in the original paper [2], using an extensible union there is no longer the need for Steele's complicated tower of data types [63].

Yet, a common thread among all the prior analyses are trade-offs between the other criteria and developer ergonomics. Usability has been one of the main criticisms of MTL-style programming. In an attempt to improve expressiveness, modularity and extensibility we have introduced several components to comprehend. We outline three consequences of this: boilerplate code is generated, there is a greater cognitive burden placed on the developer and this makes the system more error-prone through misuse.

Boilerplate. Although the term is ubiquitous in software development, and we have used it in passing a number of times already, it is instructive to clarify what we mean by *boilerplate code*. Boilerplate stems from *incidental complexity*. We informally describe this as elements in the expression of a problem domain that are extraneous; not inherent to the domain. This generates "cookie-cutter" code which contains nothing semantically interesting, instead concerning the mechanics of our programming language or framework of choice. Often it is an artefact of our tool for modular abstraction: the pattern we use for breaking apart an application into smaller components and then combining those components. A solution with no incidental complexity at all would be a DSL capturing precisely the semantics of the domain: the "ultimate abstraction" [54]. Of course, this is not practically feasible; some boilerplate is inevitable

when translating a problem domain to code. In practice, complaints about boilerplate are usually correlated with verbosity. Over the years, MTL has received a torrent of such complaints.

MTL-style programming is rife with these extraneous details. The n^2 instances problem casts a shadow over many of the positives extracted from the other three criteria. In the simple example of §5.2, we have had to define 4 boilerplate instances manually, with long lists of classes to facilitate auto-derivation of the other instances. This highlights that modularity gains of programming against a polymorphic monad do not come for free: it commands more boilerplate and less concise type signatures. It is generally still a worthwhile trade-off; a view shared by many in the pure functional Scala community. We use an abstract type constructor, typically `F[_]`, along with contextual abstractions for dependency injection [86]. In §6.4.1 we show how verbosity can be minimised, using a novel technique to make this pattern more appealing.

Another source of boilerplate is the use of `IdentityT` simply to derive `MonadTrans`. This allows `SpaceStationT` to be composed with effects that provide an instance for arbitrary transformers, disregarding conventional wisdom. In real world systems, the number of custom capabilities tends to dwarf the number of standard effects. Therefore, it is clear why defining such overlapping instances is tempting: to reduce the amount of boilerplate we are manually exposed to. This is symptom of the more general criticism that the recipe for integrating new effects into a program is convoluted. The only steps truly relevant to the problem domain are 1 and 3: the interface and its implementation which calls the ISS API. Everything else is extraneous. In §6.3 we eliminate several of these steps by applying the *tagless final pattern* of 6.1. Two different extension recipes are enumerated in §6.4.2, giving rise to a more compelling MTL pattern with respect to modularity, extensibility and ergonomics.

Cognitive Overhead. It can be argued that even worse than the boilerplate is the cognitive burden placed on the developer. When implementing interfaces in terms of class constraints, this problem is compounded. The derivation machinery fails in certain circumstances, for reasons that are not always immediately obvious. The is exemplified by `RealTimeT` in §5.3.1.

There are several options to understand in the derivation automation process, and this is not conducive to good usability. We might be tempted to declare an instance of `MonadSpaceStation` for arbitrary transformers: any `t` that implements `MonadTrans`. This controversially introduces overlapping instances. We have also seen *default implementations* used by `MonadLogger`, meaning the instance definition is one-line and in some cases avoids them altogether through language extensions [87]. This is a less contentious automation technique and is particularly useful for library authors to reduce the burden on users. Both approaches are demonstrated in C.2. They are still dependent on externalities such as transformers implementing `MonadTrans`; `RealTimeT` shows that we cannot rely on this. Having different mechanisms can generate further confusion: it may not be clear exactly which one is being triggered.

Troubleshooting derivation problems is cumbersome and different orderings will produce different compiler errors. The situation is even thornier for effects with non-standard semantics, such as state, because misunderstanding the process can lead to undesired program behaviour. Of course, we can admit defeat and define the instance in terms of the concrete `App` with anti-modular explicit lifting. Whatever solution we land on, it is suboptimal.

In the section on expressiveness we discussed the problem of expressing multiple instances of the same effect class. This can necessitate concrete transformers and explicit lifting which gives rise to further usability issues. One must carefully lift operations into the correct transformer; otherwise we might produce a program which compiles but is incorrect. Moreover, the prevalence of errors will increase in the presence of other effect interfaces since several transformers in the stack may define an instance for the same interface; as in the example given by Kiselyov et al. [22]. In the case of an effect such as state, we may opt instead to combine different stateful elements into one data structure. For example, since we cannot have a `MonadState Foo` and a `MonadState Bar`, instead we can combine them in a single `MonadState FooBar`. Again this is unsatisfactory: we lose local reasoning since an application level `StateT` is analogous to global state in imperative languages which is usually discouraged.

Susceptibility to Errors. Given all the elements of MTL, we have seen several situations where it is easy to misuse transformers. This analysis has touched upon a few of these:

1. Accidentally using an operation at the wrong level of a stack for interfaces implemented by multiple transformers.
2. Assuming standard compositional semantics when two effects interact in surprising ways.
3. Misunderstanding derivation mechanisms, causing errors or redundant boilerplate code.
4. Defining orphan instances can have disastrous consequences: the runtime behaviour of our system could be altered by a change to a module defined far away from it.

All of these can produce unintentional program behaviour. Overall, the ergonomics are at best unsatisfactory and at worst dangerous. Of our four criteria, usability is arguably the most important to software engineers in practice. Therefore, it may be tempting to admit defeat and use manual methods of effect composition and a concrete effect monad, as in example §3.3. However, we can circumvent many of these problems through appropriate design decisions.

5.4 Compendium

We distil the above commentary down to key positives and negatives of MTL-style programming. This will be useful as a high-level reference; we do the same for subsequent chapters.

Positives We summarise the benefits of MTL-style programming using the criteria sets as a guide. Particular emphasis is put on advantages over previous approaches.

A1. *Environment*. `MonadReader` and `ReaderT` supply the syntax and semantics for dynamic binding. Together they form a building block which can be composed with most classes of effects with minimal complications.

A2. *State*. `MonadState` and `StateT` supply the syntax and semantics for stateful operations. It can be composed with *some* effects without semantic implications.

A3. *Errors*. `MonadError` and `ExceptT` supply the syntax and semantics for raising and handling errors. This is useful for modelling domains with more granular exceptional situations in a type safe manner.

A4. *I/O*. Using an effect monad such as `IO` at the base of a transformer stack, I/O can be combined with other effects in a more consistent manner than the example of §3.3. `MonadIO` facilitates manual embedding of `IO` effects into a stack using `liftIO`.

B1. *Expressiveness*. We are able to express the majority of useful applications by composing the monad transformer building blocks. More challenging effects such as continuations and nondeterminism can be modelled using `ContT` and `ListT`, a notable improvement over the early approaches outlined in §4.2. Compositional semantics can be controlled through the ordering of these interpreters.

B2. *Modularity*. Using effect interfaces and automatic lifting, adding effects to programs written in terms of concrete transformers or an abstract monad do not necessitate changes to existing effectful operations. Thus, we have a viable solution for the *row* dimension of the expression problem: the DSL.

B3. *Extensibility*. We have recipes both for defining custom capabilities and adding effects to an existing interpreter. Derivation can be automated and instances are provided by libraries for the standard effects, helping this endeavour.

B4. *Ergonomics*. The system as a whole is an improvement on programming with a monolithic effect monad of §3.3 or the early monadic composition frameworks of §4.2. We have a clear pattern for writing useful software applications in a pure functional language. A notation for monad comprehension, such as `do` in Haskell or `for` in Scala, can now be used in the presence of multiple classes of effects. Using an abstract monad and type classes, we have an appealing form of modular abstraction and capability passing.

Negatives The following points, associated with subsets of **A1–A4** and **B1–B4** , will be used as a reference as we address them in Chapters 6 and 7.

C1 *Environment & Modularity*. Modifying the concrete environment type necessitates changes to all functions operating on that type using the reader effect.

C2 *State, I/O & Expressiveness*. Transformers do not provide an adequate solution for expressing concurrent state when concurrency is reified in `IO`, as is customary in practice.

C3 *Errors & I/O*. When using `IO` as the base monad, representing errors with transformers introduces a second error channel. This is at best superfluous, and at worst dangerous.

C4 *Expressiveness*. Using the ordering of transformer layers as the mechanism for compositional semantics does not have maximal flexibility. We cannot use multiple instances of the same effect without concrete transformers and explicit lifting. It is difficult to express programs that require varying semantics or interleaved instances of the same effect.

C5 *Modularity & Extensibility*. Modifying a transformer stack by adding, removing or re-ordering transformers can require either new (potentially orphan) instances or changes to existing instances for *any* transformer / effect interface pair in the stack. This constitutes an infringement on the *column* dimension of Wadler's expression problem statement, that is: the *interpretation*.

C6 *Modularity, Extensibility & Ergonomics*. The n^2 instances problem: when automating class derivation, the derivation machinery requires us to define in the order of $O(N*M)$ instances for N interfaces and M transformers. It also produces long lists of derived classes. This is both a mechanical and cognitive strain.

C7 *Modularity, Extensibility & Ergonomics*. When *not* automating derivation to avoid the $O(N * M)$ problem, one must manually lift all operations based on the position in the stack of the corresponding transformer. This is cumbersome and non-modular.

C8 *Extensibility & Ergonomics*. The recipe for defining new effects is convoluted.

C9 *Ergonomics*. It is easy to misuse transformers, compromising both the correctness and conciseness of programs.

C10 *Ergonomics*. Troubleshooting instance derivation issues is not simple, because any transformer could be preventing the effect interface from reaching the top of the stack.

Broadly, we see that usability is inversely correlated to modularity and extensibility. Programs written using concrete transformers are easier to understand for the human and the compiler; at the expense of encapsulation. Explicit lifting is cumbersome, but usually simpler both to get right initially and troubleshoot later. Conversely, abstract monads and type class constraints aid modularity, extensibility and ergonomics within *effectful programs*. Instead, complexity is pushed to the *interpretation*; giving rise to the n^2 instances problem.

Some of these criticisms are inherent to transformers, while others are an artefact of the specific formulations of Liang et al. [2] and particular language devices such as Haskell's derivation mechanisms. Many of these can be avoided by applying monad transformers more judiciously, giving rise to a practically useful programming pattern. We shall see this in the improved example of §6.3.

One of the main goals of this project was to evaluate, from first principles, effect systems based on initial DSL embeddings, free monadic or otherwise, against those based on transformers. We shall see in Sections 7.1.2 and 7.3 that frameworks built upon these DSLs are compelling alternative approaches which have the potential to improve upon transformers in almost every dimension of our criteria sets.

6 Conventional Wisdom in Pure Functional Programming

The naive programming pattern deployed in Chapter 5 constitutes a straw man, aggregating a number of bad practices to demonstrate the fallibility of MTL-style programming. However, it is only fair that we assess MTL on its best day, not its worst. To that end, we now alleviate many of the criticisms enumerated in §5.4. Conventional wisdom gleaned from the programming community, academic advances and personal experience are applied to improve upon the example of §5.2. We continue to follow the guiding principle that we should avoid becoming too wedded to any particular pattern, and instead evaluate our available tools from an objective stance.

It may be instructive to first clarify exactly what we mean by a *design pattern*. Broadly, it constitutes a recipe applicable to a common class of problems faced in software development. The extension recipe of §5.3.2 is a design pattern for extending a transformer stack with a custom effect; though one with significant drawbacks. The refactoring of concrete transformers to type class constraints constitutes another pattern, used to aid modularity. Patterns typically arise from repeatedly using a tool in practice, gradually iterating towards more satisfactory solutions that improve ergonomics, minimise incidental complexity and ameliorate common pitfalls.

With this in mind, we give the caveat that the following patterns, along with their numerous variants, can be accepted or declined depending on the circumstances. Although academic formulations, such as the expression problem [60], are helpful to give us models in which we can assess the merits of programming languages and patterns, there tends to be no silver bullet in the context of real world software. Indeed, this is the essence of conventional wisdom in software engineering: recognising when an established technique is appropriate, and when a pragmatic, perhaps unadvisable, domain-specific solution should be preferred.

We begin by presenting two patterns: *tagless final* and *the ReaderT pattern*. These are applied to our example in §6.3 and evaluated in §6.4 against our criteria sets of §2.2.1 and §4.1.

6.1 The Tagless Final Pattern

In §2.2.2 we introduced the notion of an *embedded DSL*, in particular represented as a sum type—a union of data constructors. §3.4 took this further, motivating the preference for *DSL-driven development* in software engineering contexts. We will now present the academic origins of this topic as a precursor to the pervasive *tagless final pattern*.

The ability to represent DSLs in an extensible manner was precisely the problem statement of the expression problem [60]. The example used by Wadler was a basic DSL for arithmetic operations borrowed from Hutton [88]. In the intervening years, this has become the de facto expression language used for exploring programming concepts and demonstrating specific syntax. This minimal definition allows for unobstructed scrutiny of language constructs; hence, in the spirit of Occam's Razor, this has since been dubbed *Hutton's razor*.

As is tradition, we use this example as a starting point to illuminate the different flavours of DSL embeddings. In particular, we distinguish between *initial tagged*, *initial tagless* and *final tagless* embeddings. To avoid extraneous details, the following exposition covers the topic to a depth sufficient as background for our applications of Sections 6.3, 7.1.2 and 7.3 and their accompanying analyses. A more comprehensive account of DSL embeddings is included in Appendix C.3 with interleaved commentary and examples. Kiselyov provides a rigorous academic resource [89]: we reuse his terminology in order to delineate between the various concepts.

Initial Tagged Embedding. The most basic form of Hutton's razor is a rudimentary arithmetic DSL with integer literals and addition. We call this an *object language* embedded within the host, or *metalanguage*—Haskell.

```haskell
data BasicExp = BasicI Int | BasicAdd BasicExp BasicExp

evalBasic :: BasicExp -> Int
evalBasic (BasicI i)     = i
evalBasic (BasicAdd a b) = evalBasic a + evalBasic b
```

The object language is interpreted into the metalanguage with `evalBasic`. This reduces addition of object terms down to addition of values of type `Int` in Haskell. We extend the basic DSL by adding instructions for *if-else* expressions and equality.

```haskell
data Exp = I Int
  | Add Exp Exp
  | Eq Exp Exp
  | IfElse Exp Exp Exp

data Result = RI Int | RB Bool deriving Show
```

```haskell
-- Valid expressions
ex1 = Add (I 3) (I 4)
ex2 = Add ex1 (I 5)
ex3 = Eq (I 2) (I 7)
ex4 = IfElse ex3 ex1 (I 5)
-- Invalid expressions
ex5 = IfElse (I 1) ex1 ex2
ex6 = IfElse ex3 ex1 ex3
```

Examples `ex1`–`ex4` constitute *valid* expressions; adding and comparing integer literals. These gradually build up more complex expressions from primitive constructors. Unfortunately, it is also possible to compile expressions that are *invalid*: `ex5` and `ex6`. `ex5` uses an integer literal as the discriminator when it should be a *boolean-like* expression,

whereas `ex6` returns expressions of different "types". These vagaries are symptomatic of the underlying problem: our DSL is not well-typed, detracting from its expressiveness. To remedy the issue, we need a `Result` ADT to enumerate the valid result types. This has ramifications when we try to interpret these expressions. Consider first an *evaluator*:

```
eval :: Exp -> Result
eval (I i)              = RI i
eval (Add a b)          = case (eval a, eval b) of
  (RI a', RI b')        -> RI $ a' + b'
  (_, _)                -> error "1. Cannot add non-integers"
eval (Eq a b)           = case (eval a, eval b) of
  (RI a', RI b')        -> RB $ a' == b'
  (_, _)                -> error "2. Forbid equating expressions involving booleans"
eval (IfElse disc a b) = case (eval disc, eval a, eval b) of
  (RB d, RI a', RI b') -> if d then RI a' else RI b'
  (RB d, RB a', RB b') -> if d then RB a' else RB b'
  (RB _, _, _)          -> error "3. Returned expressions must have the same type"
  (_, _, _)             -> error "4. Discriminator must be a boolean"
```

We can no longer write an interpreter with the type of `evalBasic` for two reasons. It is now possible for evaluation to fail; that is, the function must be *partial*. Therefore, to preserve purity we would have to consider returning a `Maybe` to represent failure. The alternative is to ensure that expressions are validated before being evaluated. Regrettably, that any validation has taken place is opaque in the type signature. In the spirit of favouring parsing over validation, this is unsatisfactory [90].

Secondly, our target value could now be a `Bool` *or* an `Int`. This forces us to use `Result` to *untag* types in the case discriminator, and then *tag* the return values. There are four branches of the pattern match that require tagging; we give the reason in error messages.

We *can* define a *total* pretty printing evaluator—one that cannot fail (`view` in C.3.1). This is possible because the target type `String` is static. The snag is that this interpreter would willingly print invalid expressions:

```
*DSLEmbeddings> eval ex1
RI 7
*DSLEmbeddings> view ex5
"if 1 then (3 + 4) else ((3 + 4) + 5)"
```

Initial Tagless Embedding. These problems of tagging and partial evaluation can *both* be resolved by incorporating better type-safety in the DSL. One means of achieving this is using a *generalised algebraic data type* (GADT). `ExpIT` contains consistently-named data constructors:

```
data ExpIT a where
  I'      :: Int -> ExpIT Int
  Add'    :: ExpIT Int -> ExpIT Int -> ExpIT Int
  Eq'     :: ExpIT Int -> ExpIT Int -> ExpIT Bool
  IfElse' :: ExpIT Bool -> ExpIT b -> ExpIT b -> ExpIT b

-- Valid expressions
ex7 = Add' (I' 4) (I' 5)
ex8 = Eq' (I' 4) (I' 5)
ex9 = IfElse' ex8 ex7 (I' 6)
-- Invalid expressions no longer compile, a win!
-- ex10 = Add' (B' True) (I' 4)
-- ex11 = IfElse' ex8 (B' True) (I' 2)
```

The polymorphism of `a`, the characteristic feature of a *generalised* ADT, allows us to assert the three object language invariants at compile time. Now our interpreters are *total and well-typed*:

```
evalIT :: Eq a => ExpIT a -> a
evalIT (I' i)              = i
evalIT (Add' a b)          = evalIT a + evalIT b
evalIT (Eq' a b)           = evalIT a == evalIT b
evalIT (IfElse' disc a b) = if evalIT disc then evalIT a else evalIT b
```

`Eq` here is the standard Haskell type class for equality; an element of the metalanguage. GADTs are a useful tool for typed eDSLs and they form the basis of extensible effect systems [22, 23]. Another advantage of this approach is that it is more clear which types are intrinsic to the DSL and which can be abstracted away since they are arbitrary. Thus, initial embeddings have been described as *context free grammars*, since they constitute a set of instructions free from any semantic domain [89].

However, `ExpIT` suffers from a different problem: it lacks extensibility. Consider attempting to extend our DSL with integer multiplication:

```
data ExpIT a where
  --- other constructors...
  Mult :: ExpIT Int -> ExpIT Int -> ExpIT Int
```

This adds an instruction—a *row* in Wadler's characterisation [60]—to our instruction set; forcing changes to all existing interpreters to satisfy exhaustive pattern matching. This is undesirable and reflects that this DSL construction violates the expression problem.

Final Tagless Embedding. Using a different tagless style we can solve the problems of *tagging*, *type-safety* and *extensibility* in one fell swoop. A *final embedding* is characterised by a parameterisation of the DSL over the interpretation. Instead of a data type, we declare classes to represent the two DSLs:

```
class AddFT a where
  int :: Int -> a
  add :: a -> a -> a
```

```
class EqFT a where
  eq     :: Int -> Int -> a
  ifElse :: Bool -> a -> a -> a
```

This is no longer context free: denotations of object terms are defined directly, though abstractly, in terms of host language types. Thus, the static guarantees of our DSL are inherited from the metalanguage—Haskell. Examples of interpreters make this clearer:

```
instance AddFT Int where
  int     = id
  add a b = a + b
```

```
instance EqFT Bool where
  eq             = (==)
  ifElse disc a b = if disc then a else b
```

Interpreters in this embedding take the form of *type class instances*. Note that type classes are merely one—particularly useful—representation of a final DSL in a host language. An alternative approach involving OCaml *modules* was described in the inceptive paper of Carette et al. [91]. We can write expressions using these class operations as language terms without assigning any semantics:

```
-- ex12 :: AddFT a => a
ex12 = add (int 3) (int 5)
ex13 = add ex12 (int 2)
ex14 = eq ex12 ex13
ex15 = ifElse ex14 ex12 ex13
```

```
*DSLEmbeddings> ex12 :: Int
8
*DSLEmbeddings> ex15 :: String
"if False then (3 + 5) else ((3 + 5) + 2)"
```

These examples are inferred as having polymorphic types, as the commented-out `ex12` signature illustrates. This type reads as: any `a` in which integers can be embedded, also supporting some notion of addition. During interpretation the type is inferred based on the *typed context* as shown by the REPL output above. `ex15` betrays one of the characteristics of a final embedding: the expressions are *reduced* to their representation in the host language. This precludes opportunities for transformation in the object language, such as optimisations based on known reduction rules within our domain. It is possible to regain this lost power by interpreting a final embedding back into an initial embedding. This has applications in program optimisation, static analysis and testing [92].

Notably, we can now extend the DSL with multiplication without demanding recompilation of existing interpreters. We simply define a new type class and instances:

```
class MultFT a where
  mult :: a -> a -> a
```

```
instance MultFT Int where
  mult = (*)
```

Finally, some closing examples which combine every operation of our tagless final eDSLs:

```
ex16 = add (mult (int 4) (int 2)) (int 1)
ex17 = mult (int 3) (int 3)
ex18 = ifElse (eq ex16 ex17) (int 50) (int 51)
```

```
*DSLEmbeddings> ex16 :: Int
9
*DSLEmbeddings> eq ex17 ex16 :: Bool
True
*DSLEmbeddings> ex18 :: String
"if True then 50 else 51"
```

Effect Systems. How does all of this relate to our practical study of effect systems? The tagless final embedding should be familiar from the constructor classes of §5.2, for instance:

```
class Monad m => MonadSpaceStation m where
  fetchLocation :: m Location
```

More generally, MTL-style type classes are an example of a tagless final DSL encoding. We can now view the type parameter `m` in a new light: this captures that the DSL is polymorphic over the semantic domain, which in this case happens to be a monad. Haskell programmers have been applying this technique for at least a couple of decades; before the dots were connected in the original 2007 paper [91]. Hence, the *mtl* library can be viewed as a collection of tagless

final DSLs for the standard effects. Our key adjustment is to bypass custom transformers such as `SpaceStationT` and instantiate this interface, or rather this *tagless final DSL / algebra*, directly for the application interpreter. Since this custom effect is situated within a stack, we already implemented the interface in terms of class constraints. Consequently, it is trivial to make this modification; as we illustrate in §6.3.

Tagless final has gained significant adoption in the pure functional Scala community over the last few years. With fewer ties to academia and programming language theory, in this context the pattern has shed its original characterisation to become simply an appealing form of modular abstraction. A Scala example applying this technique to the ISS problem is in Appendix B.2. This will be presented in §6.4.1 and reflected upon throughout the analysis.

6.2 The ReaderT Pattern

In direct contrast to the previous section, *the ReaderT pattern*—coined by Snoyman in a 2017 blog post [76]—emerged from real world application of pure functional programming. It has since gained favour in the Haskell community, with frameworks developed to aid its adoption [93, 94, 95]. Pure functional Scala developers make similarly motivated design decisions, leading to a comparable pattern. Key differences will be highlighted in §6.4.1.

This is not an entirely well-defined pattern, but rather a set of recommendations for avoiding common pitfalls when programming with transformers. It also drastically improves usability by removing layers of indirection wherever possible. There is no longer the monstrous monad transformer stack and resulting cognitive overhead described in §5.3.2. Conveniently, these recommendations form natural pairings with certain issues enumerated in the compendium of §5.4. Therefore, we use these points as a guide.

***C1**. Nonmodular Concrete Environment.* The concrete `Config` environment damages modularity: changing the structure of our environment can break functions operating on this data type anywhere in the codebase. Moreover, it is unclear from function signatures which elements of the config are necessary; this could be made explicit but would require manipulating the environment locally or passing parameters manually. Therefore, it would be preferable to abstract constituents of the environment in a manner comparable to the abstraction of the monad in §5.3.2. Our approach is similar: we use types classes to abstract the concrete environment and specify only the relevant parts.

```
class Has a t where
  getter :: t -> a

readEnv :: (MonadReader t m, Has a t) => m a
readEnv = reader getter
```

We use `Has` to *project* out of the environment type. In conjunction with `MonadReader`, we provide a convenient projection function `readEnv` whose type can be inferred from the context. This induces a simple and modular pattern which we use extensively in the §6.3 implementation. Addressing our problem statement explicitly: now if we modify the environment type, only the instances of `Has` used to project specific modified records must be changed. Importantly, effectful programs—our business logic—remains untouched. This is true regardless of whether we use a concrete `ReaderT` or class constraints.

***C2**. Concurrent State.* We gave an example of `StateT` interacting in confusing ways with `IO` in §5.3.2. More generally, using transformers that mutate state like `StateT` or `WriterT` alongside control effects such as errors, concurrency and continuations can produce surprising results. The semantics are difficult to comprehend and having `IO` at the base further muddies these waters.

Fundamentally, if state transformers are used in large subsets of your application, then the ability to reason about state is no better than mutating global variables in imperative languages. Consider calling `put` in this situation: this sends the state somewhere potentially very *far away*. This may be modified somewhere else also far away. In order to understand the order of state modifications, one needs to look at every instance of `put` or `modify`, mentally tracking their relative positions and the behaviour to expect as a result. This is wholly worse than *setters* in OOP classes which at least have a well-defined encapsulation boundary.

Snoyman best articulates the solution: "may as well call a spade a spade, and accept that you have a mutable variable" [76]. In particular, a purely functional mutable variable is, like `IO`, a data structure abstracting concurrent state through a set of operations. These emulate imperative mutability and are comparable with *volatile* variables. We outline the different flavours of concurrent state primitives in §6.4.1. Since these operations constitute *observable effects*, to retain purity we must capture their invocation using our usual tool for delaying computations: `IO`. Mutable references such as these are designed with concurrency in mind; the exact mechanism is abstracted behind `IO`. How do we incorporate mutable references into our program? Since they are simply values, we may choose to pass them in our environment. Alternatively, we might use abstractions to hide the `IO`. We demonstrate an `MVar` in practice in Appendix A.6 with reflection in §6.4.2.

***C3**. Two Error Channels.* The approaches of both §3.3 and §5.2 fell into the trap of mishandling errors embedded in `IO`. We can alleviate this problem with a simple recommendation: avoid error transformers such as `ExceptT` in the presence of `IO`. The type-safety of `MonadError` is still appealing and we can preserve these benefits using custom exceptions. The problem of composing state and errors described first in §3.1.2 is re-examined in §6.4.1.

What are the top-line consequences of applying these adjustments? Immediately we have eliminated `ExceptT` and `StateT`. Our monad transformer stack for the majority of real world applications becomes a variation of `ReaderT Env IO`: far removed from our complex `App` interpreter. Combining these design decisions with a tagless final approach for custom effects, we naturally avoid many of the other issues relating to modularity, extensibility and ergonomics. The extent of the n^2 instances problem, the ensuing usability issues and performance overhead grows with the number of transformers in our stack. Hence, by virtue of deploying only one transformer—and one with standard interaction semantics—we mitigate most negatives of §5.4. A more precise analysis of the ramifications is given in §6.4.

6.3 Example Implementation: Monad Transformers, Pragmatically

Combining these two techniques with a healthy dose of pragmatism, we reimplement the example of §3.2. This produces a more compelling MTL-based programming style and relieves many of the concerns from §5.4. The full solution is in Appendix A.5. The most significant differences lie in our increased use of the reader effect. We begin with effect *syntax*:

```
newtype Sleeper m = Sleeper { runSleeper :: Microseconds -> m () }
-- sleep :: (MonadReader t m, Has (Sleeper m) t) => Microseconds -> m ()
sleep s = readEnv >>= (`runSleeper` s)
sleepWithDelay :: (HasSleeper r m, Has Seconds r, MonadReader r m) => m ()
sleepWithDelay = readEnv >>= (sleep . secsToMicrosecs)
```

We use a `newtype` to wrap a function for semantic blocking and the `Has` pattern of §6.2 for *projection*. Notably, the type of `sleep` is inferrable based on the context in the function body: see the commented-out signature. This syntax can now be used without knowledge of the concrete environment or interpreter:

```
type MonadAppCommon r m = (MonadReader r m, MonadRef m, MonadSpaceStation m,
                           HasLogger r m, HasSleeper r m, Has Seconds r)

trackSpaceStation :: (MonadAppCommon r m, Has Count r, MonadCatch m) => m ()
trackSpaceStation = catch program logFailure
  where
    program :: (MonadAppCommon r m, Has Count r) => m ()
    program = do
      initialCount    <- readEnv
      initialLoc      <- fetchLocation
      initialRef      <- newRef (AppData initialLoc (initialCount - 1) [])
      finalSpeeds     <- trackSpeed initialRef <&> speeds
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure (e :: ISSException) =
      log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: MonadAppCommon r m => Ref m AppData -> m AppData
trackSpeed ref = do
  state <- get ref
  if remainingRequests state <= 0
    then log "No more requests to send to ISS" $> state
    else do
      sleepWithDelay
      currentLoc <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      put ref newState
      trackSpeed ref
```

The unfamiliar elements are explained in the analysis. There is no mention of the specific environment containing the configuration and I/O effects. This is made concrete by embedding the functionality in the `Env` data type:

```
data Env m = Env { conf :: Config, logger :: Logger m, sleeper :: Sleeper m }
instance MonadReader (Env m) m => Has (Sleeper m) (Env m) where getter = reader sleeper
```

This combines the existing `Config` with further capabilities. `Env` needs to be parameterised by an abstract monad because it inherits the polymorphism from `Sleeper` and `Logger`. The `Has` instance projects the sleeper out of the environment and is resolved during interpretation. We make use of this pattern in our new implementation of `MonadSpaceStation`:

```
instance (HasLogger r m, Has Request r, Monad m, MonadReader r m, MonadCatch m,
          MonadIO m) => MonadSpaceStation m where
  fetchLocation = do
    request <- readEnv
    body    <- httpJSON request `catch` (throwM . ISSException)
    getResponseBody body `tapM` log
```

We see the granular environment in action: `fetchLocation` is given precisely the power it needs—only the logger and HTTP request. Importantly, the instance applies to an arbitrary monad `m`, rather than a transformer with `m` as its base monad. With the context of §6.1 we can now view the interface as a custom tagless final DSL and this instance as an interpretation. The main difference is that the *target* of the interpretation is an abstract monad, rather than `Int` and `String` from the Hutton's razor examples. This is eventually instantiated to our concrete interpreter:

```
newtype App a = App { runApp :: Env App -> IO a }
  deriving (Functor, Applicative, Monad, MonadThrow, MonadCatch,
            MonadReader (Env App), MonadIO) via ReaderT (Env App) IO
```

Notably, there are no transformers—only the `runApp` accessor. The use of `via` is syntax we have not seen: this treats `ReaderT` as an implementation detail by abstracting it away in the derivation of `MonadReader`. It is arguably overkill here, but can be useful in situations where a clean API is desirable. Our top level evaluation becomes:

```
main = let logger  = Logger (liftIO . print)
           sleeper = Sleeper (liftIO . threadDelay . getMicroseconds)
       in runApp trackSpaceStation $ Env config logger sleeper
```

The lack of a single transformer in this interpretation is a drastic departure from the large stack in §5.2. We complete our shift in perspective: MTL is a set of tagless final DSL embeddings for standard effects. We still call this *MTL-style programming*, with the acknowledgement that the *transformer* element is largely a historical artefact.

6.4 Analysis

6.4.1 Representing Effects

***A1**. Environment* Immediately we have solved **C2**: our environment is decoupled from the business logic. We are mixing two methods for dependency injection: tagless final type classes and *method dictionaries*. This was done to illustrate the advantages of both techniques over the transformer-heavy approach. In practice it is preferable to follow a consistent pattern within a codebase. Since both are relevant to environment, we contrast the approaches here.

Tagless Final Embedded DSLs. The effectful program uses four tagless final interfaces: `MonadReader`, `MonadCatch`, `MonadSpaceStation` and `MonadRef`. These provide, respectively: `reader`, `catch` / `throwM`, `fetchLocation`, and `newRef` / `readRef` / `writeRef`. The most significant benefit is that we can extend the program with new effects without having to change the *being* or the *doing*: syntax is immediately available when a class is defined and semantics are isolated to the instances. We have a compelling solution to the expression problem.

Three of these effects are provided by external libraries. It is now seamless to incorporate such interfaces without worrying about the various transformer complexities such as lifting, compositional semantics, and n^2, orphan and overlapping instances. We get the benefits of derivation automation without the drawbacks: `App` inherits every instance from `IO`, except `MonadReader` which is derived via `ReaderT`. Compositional semantics are unavoidable but not so explicit in this example—we analyse effect composition in Sections 6.4.1 and 6.4.2.

Method Dictionaries. Some simple adjustments give rise to another appealing form of dependency injection. The embedding of capabilities in `Env` is an example of a *method dictionary*, also known as a *record of functions*. In essence, we are embedding functions into a data type and providing record accessors—*keys*—to project them out. These accessors need only be used once in the `Has` type class instance for projection. Combining this approach with *lenses*—a technique for viewing and modifying nested immutable data—gives us a powerful tool for modular abstraction.

The type signatures are informative but admittedly somewhat verbose. Although the `Has` instances are less cognitively challenging than the transformer instances described throughout §5.3, they do still constitute boilerplate. This is a usability concern and will be discussed in §6.4.2.

Hybrid Languages. Pure functional languages such as Haskell, PureScript and Idris are opinionated: they contain a minimal number of language constructs to guide programmers to a particular style. This aids usability and gives us strong guarantees since referential transparency is the default. In contrast, hybrid languages such as Scala support a wide array of usage patterns by providing a larger core language with features such as OOP-style classes, contextual abstractions and nested functions. This can have disadvantages: bloated languages give rise to more opportunities for misuse and can have a higher barrier to entry.

However, Scala, and in particular Scala 3, strikes a reasonable balance in expressiveness. As a result, using frameworks such as Cats Effect [11] we are able to write programs in an style identical to this Haskell example. State, errors and I/O can be embodied by tagless final DSLs represented as *traits*. In contrast, the environment or *reader* effect pervasive in §6.3 can be expressed using entirely different mechanisms involving hybrid language features. Here we present only the snippets necessary as a basis for comparison: the full Scala 3 solution is included in Appendix B.2 and our novel dependency injection pattern is explained in §B.3.1.

A brief note on terminology. Scala inherited the notion of *OOP-class methods* from Java. The `def` keyword represents this syntactically. This is distinct from *functions*—as in *first-class function values*. We use both terms to mean *method* unless indicated otherwise.

```
// Program entry point, the IO is run in the standard JVM 'main' method
def run: IO[Unit] =
  EmberClientBuilder.default[IO].build.use { (client: Client[IO]) =>
    // Effect implementations providing semantics
    given Logger[IO]       = IO.println
    given Sleeper[IO]      = IO.sleep
    given SpaceStation[IO] = () => fetchLocationImpl(config.issRequestUri, client)
    trackSpaceStation(config)
  }

def fetchLocationImpl[F[_]: Logger: Concurrent](uri: Uri, client: Client[F]): F[Location] =
  client.expect[Location](Request(Method.GET, uri)).flatTap(log)

trait Logger[F[_]] { def log(msg: String): F[Unit] }

def log[F[_], A: Show](msg: A)(using l: Logger[F]): F[Unit] = l.log(msg.show)

def trackSpaceStation[F[_]: MonadThrow: Logger: Sleeper: SpaceStation](config: Config): F[Unit] =
  def trackSpeed(data: AppData): F[AppData] = ...
  // Equivalent to the nested 'program' seen in the Haskell examples
  val program: F[Unit] = ...
  // Equivalent to use of 'catch' in Haskell examples
  program.handleErrorWith(e => log(show"Failed to call ISS with error: $e"))
```

We briefly explain contextual abstraction features since the keywords are unique to Scala. Just like Haskell type constraints, the `given` / `using` keywords provide *implicit* passing of *context*. To satisfy the compiler, when calling a function there must exist a `given` declaration for every type present in a `using` parameter list. Then, we do not need to pass common elements of the environment to functions explicitly.

`Logger` is an example of a tagless final interface parameterised over an interpreter, as we have seen in Haskell. Unlike for the abstract monad `m`, we must explicitly mark `F` as a type constructor by giving it a "hole": `[_]`. For conciseness, `F[_]: Logger: Concurrent` is syntactic sugar for `using Logger[F], Concurrent[F]`. These are called *context bounds*. Given these language features, we can provide syntax such as `log` for use in effectful programs without needing to explicitly pass `Logger` (or `Show`). A key difference is that there are no transformers: our type constructor is instantiated directly to `IO` for every DSL.

Further differences are illustrated by the explicit passing of the `config` value. To avoid threading this through multiple functions, we use a *nested method* `trackSpeed` which lives locally within the context of `trackSpaceStation`. It *inherits* access to everything in the outer scope: the `F` type constructor, all the type classes and the `config` value.

Another way to capture context is by making values available to instances of OO-style classes. Although subtle, this is happening above: every trait is instantiated to a JVM class. Taking the example of `SpaceStation`, using traditional Java style we could write:

```
class SpaceStationImpl(config: Config, client: Client[IO])(using Logger[IO]) extends SpaceStation[IO]:
  def fetchLocation(): IO[Location] = fetchLocationImpl(config.issRequestUri, client)
```

This explicitly names the class and uses a constructor to pass the required context. Actually defining a named class like this could be considered boilerplate: it does not *need* to have name. In essence all we need is some entity capturing the `fetchLocation` operation in a manner that can be abstracted, tested and injected into other components of our application. This is precisely what is achieved by the one-line definition of `SpaceStation[IO]`. We can create *anonymous*, or unnamed, instances of traits using the `new` keyword. Here we go one step further: since the trait has only one method, we can write an anonymous *first-class function* which is automatically given as the trait's implementation. This is inferred from the type on the left-hand side. The implementations of `Logger` and `Sleeper` are yet more terse; deferring to standard `IO` operations.

Given that this pattern with concrete named classes is ubiquitous in object-oriented programming, environment passing is not often considered an effect but rather an intrinsic aspect of OOP. Here we see that it is just one means of achieving modular abstraction: the goals of dependency injection in OOP and FP are really the same, just the mechanisms differ. All together, the combination of contextual abstractions and type class instances induce an appealing pattern for real world pure functional programming in Scala.

Dependency Injection in Scala 3. One of the criticisms of tagless final is that function signatures become verbose with long lists of context bounds [96]. Our example has only four, but functions can grow unwieldy in large applications. This should generally be mitigated by better software design: long lists of dependencies indicate that a module has too much responsibility and should be broken apart. However, we may also want to group type class constraints like in Haskell. Until Scala 3 this was not possible in a noninvasive manner; we had to define traits extending different tagless final algebras. We can achieve this goal using a novel Scala 3 formulation and our example as a guide.

Scala *context functions* allow us to define functions accepting implicit context as *first-class values*; as if we had defined them as methods with a `using` clause [97]. For instance, we can rewrite the `log` function:

```
def log[F[_], A](msg: A)(using l: Logger[F]): Show[A] ?=> F[Unit] = ...
```

This is functionally identical. The return type indicates that the body requires an implicit `Show[A]` instance. This feature can be used for our tagless eDSLs too. Combining this with some type-level tricks to produce a more usable API, we can create a pattern similar to Haskell's `Has` without needing a transformer to carry the environment. Ostensibly, this is a novel approach; the Scala community has generally only been using Scala 3 in practice for a year. Real world services have been deployed to production using the pattern described here: though the API certainly needs more scrutiny. A full, verbose explanation is provided in Appendix B.3.1. Here we present the high-level summary.

We somewhat push Scala's boundaries but, notably, this is *not* meta-programming. These are all standard language features. Given sufficient knowledge of the constituent features, it is not a great leap to understand how the pattern works holistically.

```
type Has[Eff[_[_]]] = [F[_], UsedIn] =>> Eff[F] ?=> UsedIn

infix type *[First[_[_], _], Second[_[_], _]] = // Nests Second inside First
  [F[_], UsedIn] =>> First[F, Second[F, UsedIn]]

infix type ^[First[_[_], _], Second[_[_], _]] = // Delays type application
  { type Use[F[_], UsedIn] = *[First, Second][F, UsedIn] }
```

We can understand `Has` as follows: the "effect" `Eff`, using the term loosely, is used to produce a value of type `UsedIn`. This constitutes the base case. Then `*` allows us to *compose* two environments. Specifically, these environments must have the shape of tagless final DSLs: they must accept a type constructor—the `F`—and a regular type—the value produced by the function. The hat or *roof* of an environment, `^`, is an unfortunate consequence of Scala not supporting partial type application by default. To aid usability, we are forced to push type application to a *path-dependent type* named `Use`. Altogether, we apply these constructs to our example:

```
type Common     = Has[Monad] * Has[Logger] * Has[Sleeper] ^ Has[SpaceStation]
type Env[F[_]] = [UsedIn] =>> Common#Use[F, UsedIn]

def trackSpaceStation[F[_]: ApplicativeThrow](config: Config): F[Unit] Using Env[F]
def trackSpeed[F[_]](data: AppData): F[AppData] Using Env[F]
```

We must always *complete* a grouping with `^` before the final `Has` constraint. Eventually we need to apply the types to construct our environment: this is done with `#Use`. For convenience, `Using` is an *infix type* defined to avoid having to nest the return type in `Env`. It is named to have consistent semantics with the `using` keyword.

We see that `trackSpeed` does not need the ability to throw or catch errors, so it does not need `MonadThrow`—only `Monad`. `trackSpaceStation` can then extend this common environment with a single `ApplicativeThrow` context bound to allow the use of `handleErrorWith`. This is a modularity improvement. Combined with a linting tool to flag to the user that a particular type class is unused, this gives rise to a powerful pattern for modular functional programming.

The result is less verbose than declaring several type class constraints, and this benefit would grow with the number of type classes and functions. Therefore, the cost of defining these groups will be amortised with the project size. Moreover, the original example can be refactored to this pattern with *no invasive changes*: the compiled artefact is roughly the same. Since we have an alternative to nested functions and local methods of classes, another benefit is that we are encouraged to break modules apart into smaller components. Having more granular function definitions improves maintainability since it is easier to identify duplication within a codebase. Smaller functions are also easier to test: it is not possible to test nested or class-local methods since they only exist within the scope of their parent.

In practice, it is typical to use powerful effects, such as `Async` for asynchronous and concurrent computations, at the *edges* of a backend service; in interactions with databases, web APIs or messaging systems. Conversely, at the core of services—the business logic—there are often less powerful constraints: `MonadThrow`, `Logger`, `Metrics`, `Parallel` in addition to domain-specific abstractions. Identifying common groupings provides more motivation for a mechanism similar to that available in Haskell. Here we have such a mechanism.

The biggest criticism is usability. There are a number of advanced language features involved: partial type applications, type lambdas, path dependent types and even context functions are not yet widely known. The tagless final pattern already has its detractors with respect to teachability and this may not aid its defence in that regard. Forcing users to call `#Use` is unsatisfactory. However, the framework itself—`Has`, `*`, `^`, `Using`—need only be written once and could be pushed to a library. Given a set of example use cases, users would not need to understand every element in order for the pattern to be useful.

Overall, this is another option for dependency injection. We have applied it to tagless final DSLs in particular, though it can be easily extended to other contexts. Moreover, it may be used in conjunction with other tools for contextual abstraction such as context bounds and regular `using` clauses. As has been a common theme throughout: we can take it or leave it based on the circumstances as well as personal preference.

We now have several techniques for injecting both data and capabilities deep within an application in purely functional and hybrid languages. Most of the remaining disadvantages concern ergonomics and will be examined in §6.4.2.

A2. *State* We have applied the advice of §6.2 with additional syntax for consistency with `StateT`:

```
get :: MonadRef m => Ref m AppData -> m AppData
put :: MonadRef m => Ref m AppData -> AppData -> m ()
```

`MonadRef` is another tagless final interface, abstracting away operations on mutable references: here `IORef`. Our `App` interpreter inherits its operations from `IO`. Note that *creating* a mutable value is also considered an observable effect, since it entails the reservation of a memory cell.

Pure Functional Mutable References. In imperative programming languages the distinction between different flavours of mutable state is not always obvious. We enumerate the standard use cases and give the pure functional equivalent:

1. A variable which may be initially empty is typically declared using a `var` and assigned to `null` in imperative languages. In such a situation we can use an `MVar a`: a concurrency primitive which may or may not contain a value of type `a`.

2. If instead we want the value to only be settable *once*, perhaps with multiple threads competing to do so, we may opt for `cats.effect.Deferred`.

3. Often we already have an initial state and need to perform atomic reads and writes—a Haskell `IORef` or `cats.effect.Ref` is applicable here.

Less frequently, we require complex state modifications for which ordering is critical and *rollbacks* can be performed: a transactional semantics. This is an in-memory analogue of transactions in a relational database. Imperative languages use syntax such as `volatile` to mark that a particular variable must support concurrent modification. Larger *atomic* values, mutexes and semaphores can be built from such primitives. In pure FP, the primitives listed above provide operations to support this endeavour by applying synchronisation barriers to achieve atomicity. However, it may be more appropriate to use comprehensive APIs provided by *software transactional memory* frameworks [98, 99]. These were inspired by the original work of Peyton Jones [100].

We now have an abundance of solutions to the problem of **C2**: mutable references can be used to support concurrent and global state if necessary. This is more flexible than transformers since `StateT` is not suited to the first two situations. With no initial state we would need to represent optionality in the state type parameter `s`—generating incidental complexity.

State in Real World Applications. The example of §6.3 and Appendix A.5 conspires to include purely functional mutable references for illustration. In reality, we would simply use recursion as is done in the Scala example of §B.2. More broadly, mutable state should be used judiciously. Allowing unconstrained mutation by pushing state modifications into the `IO` monad damages local reasoning. If it is truly necessary to reach for mutable references, we can at least temper their damage by restricting them to small subsets of our application or encapsulating them behind abstractions. We demonstrate this in §6.4.2. Method dictionaries and `Has` is another solution; providing more granular access to the mutable references embedded in our environment.

We can solve the majority of real world problems without mutation using standard techniques such as recursion, lenses, folds and other combinators. In Scala or OCaml, library authors might opt for constrained local mutation. Even in Haskell, `unsafePerformIO` is used carefully in the library ecosystem. In application development, this is rarely necessary. The recommendation to use mutable references is primarily for situations involving nondeterminism or errors. Conversely, state monad transformers can be particularly useful when the base monad is *not* `IO` .

For modern backend services, it is generally assumed that a program will be be scaled across multiple instances. Consequently, it is best to opt for solutions which scale to an arbitrary number of servers. Relying on in-memory state should be avoided since it is local to a particular instance. This does not lend itself to robust and fault-tolerant systems. Instead, we should favour external dependencies which can be shared across instances.

A3. *Errors* By pushing all errors into the `IO` monad we have addressed **C3**: there is only one error channel. All ISS API call failures are handled consistently. Raising errors is encapsulated by the `MonadThrow` interface and this is extended with handling by `MonadCatch`. Hence, we retain the modularity benefits of `trackSpaceStation'` in §5.3.2 with its `MonadError` constraint. We discuss other implications of this approach.

Composing State and Errors. Applying our recommendations, how does the composition of state and errors compare with §3.1.2? Both errors and stateful modifications are now being pushed into the `IO` monad. Therefore, compositional semantics is reduced to the monadic composition of `IO` values. We contrast this against the other frameworks in §7.4.2. Conceptually, this is more comprehensible than a stack of transformers and the resulting interaction complexities. Recalling that monads are the equivalent of the semicolon operator in a pure functional context [58], the semantics are consistent with the imperative equivalent: state is preserved. `StateT` over `ExceptT` or `MaybeT` shares these semantics and is *structurally* equivalent to the first ordering we gave: `Int -> (Int, Maybe a)`.

Typed Errors. In §5.3.1 we motivated the preference for using precise errors when possible. Here we have followed this advice as best we can, wrapping the most generic exception `SomeException` in our own type `ISSException`. This occurs at the boundary of our program and the external library: first we `catch` the exception, wrap it in our custom type and re-throw with `throwM`. In `trackSpaceStation`, the type of exception to catch is inferred from `logFailure`. Adding more granularity would require coupling our program to specific errors propagated by *http-conduit*. This has questionable modularity and would be cumbersome to sustain throughout a large codebase. Instead, we make clear the source of the exception and allow our application to handle it appropriately. In Scala this is more critical because we often use Java libraries designed for consumption by imperative programs. These can throw any errors of type `Throwable`: the root of the exception hierarchy. Therefore, we treat it as an FFI by suspending method calls in an effect monad to regain referential transparency.

It is now more explicit which exceptional situations are expected—*known unknown*—exceptions, and which are software errors. The former is captured by `ISSException`. For the latter, inserting `log (error "Fake" :: String)` into the body of `program` produces an unhandled exception. This constitutes an *unknown unknown* error and indicates that our application has a bug which a programmer will need to address.

***A4**. I/O* The abstractions of I/O operations have been covered extensively in the environment discussion of §6.4.1. We see that the `IO` monad has broader applications than just I/O: we use it for state transformations, error propagation and as the interpreter for tagless final eDSLs. This is not necessarily a good thing.

6.4.2 Composing Effects

***B1**. Expressiveness* We have addressed the most urgent practical problems of monad transformers with respect to expressiveness. Our new techniques provide a toolkit for solving the majority of real world problems in a maintainable way. There is now space to extend our remit to less pressing concerns.

Compositional Semantics. The incidental complexity of the largely irrelevant transformer ordering has been avoided. This is a consequence of two adjustments: discarding transformers for custom effects and avoiding standard transformers. Composing state and errors now has fixed state-preserving semantics; verified in Appendix C.5. The lack of flexibility caused by static ordering does not present in our simple example, but remains a problem inherent to transformers. This was captured by **C4** and discussed in §5.3.2.

Higher-order and Control Effects. We outline some special classes of effects which are particularly challenging to express using transformers. Most intricacies of the many proposed solutions and caveats are considered out of scope: we attempt to keep focus on the specific implications for writing applications using MTL-style.

The two examples used by Kiselyov et al. to demonstrate the limitations in expressiveness of transformers have something in common [22]. They involve *higher order* and *control* effects. These include three operations native to MTL effects we have made use of: `throwM` and `catch` of `MonadCatch`, and `local` of `MonadReader`. Error effects are the most intuitive example of a control effect: they modify control flow through *short-circuiting* semantics. `local` is part of the minimal definition of `MonadReader`, enabling local environment modification. For reference:

```
throwM :: Exception e => e -> m a
catch :: Exception e => m a -> (e -> m a) -> m a
local :: MonadReader r m => (r -> r) -> m a -> m a
```

Recall also `concurrently` of §5.3.2.

```
concurrently :: IO a -> IO b -> IO (a, b)
```

Although `throwM` modifies control flow, we can divorce it from the others because it is a *first-order effect*. Specifically, it does not *operate on* monadic values. Moreover, notice that there are *m*-computations in a *negative position* in the three other operations. Namely: the first argument of `catch`, the second argument of `local`, and both arguments of `concurrently`. We need to provide a monadic value to produce a result. This structure has ramifications. Consider trying to lift `local` through another transformer, say `SpaceStationT`:

```
localSST :: MonadReader r m => (r -> r) -> SpaceStationT m a -> SpaceStationT m a
localSST f sst = SpaceStationT $ IdentityT (local f (runIdentityT $ runSpaceStationT sst))
```

We are forced to unwrap the transformer, producing an `m a`, apply `local` to the underlying monad, then wrap it back up in our transformer. Notably, *it is impossible to write this function using* `lift`. In principle, we should be able to lift `local` through any transformer. This is symptomatic of `MonadTrans` fundamentally lacking power. Specifically, it does not capture the *monadic state* of the lifted operations. This is distinct from the *state monad*: we are referring to the more general notion of state carried by *any* monadic effect. For instance, the monadic state of `MaybeT` is that the monadic value may be optional, and the monadic state of `StateT` is propagation of the state value in a tuple alongside the monadic value. By abstracting this monadic state, we should be able to describe a stricter interface than `MonadTrans` which would allow operations such as `local` to be lifted through arbitrary transformers.

Solutions to this problem have been embodied by abstractions such as `MonadTransControl` of the *monad-control* Haskell package [101]. This allows a wide number of standard and advanced higher-order effects to be lifted through *most* transformers. For example, `local` and `throwM` can now be lifted through all standard transformers.

A closely related problem can be understood by attempting to lift `concurrently` into some arbitrary monad. In the naive example of §5.2 we used `liftIO` to lift *`IO`-values* to the top of our large stack of transformers. This was possible due to `App` inheriting the `MonadIO` class from `IO` at its base. However, this is insufficient to lift a function with monadic values in a *negative*, or *input*, position. Moreover, `liftIO` is crude: in theory we should have a more general operation for lifting programs in an arbitrary base monad into a transformer stack. Then `MonadIO` would be simply a *specialisation* of this operation to `IO`.

This proves to be even thornier than lifting control effects through single transformers. A technical blog post by King gives a helpful and comprehensive exposition of this problem from first principles [102]. To summarise, whereas `MonadTransControl` allows control operations to be lifted through transformers generically, `MonadBase` and `MonadBaseControl` allow us to lift operations through *whole stacks* of transformers. Thus, the state of multiple monadic effects must be captured, *unlifted*—or *run*—and restored. Unfortunately there are limitations and dangers to this approach. We might reasonably expect a function such as `IO Int -> IO Int` to be liftable into a transformer stack. It is not: we require the function to be polymorphic in its monadic value. More worringly, it is possible to discard monadic state if we're not careful, producing counterintuitive semantics. This is compounded in the presence of concurrency. For example, we have seen in §6.4.1 that pure mutable references such as `IORef` preserve state when exceptions are thrown. These semantics would be overridden when lifting `catch`: if an exception is thrown, state changes would be *rewound* since only one of the input program and the exception handler are run. This is inconsistent with our understanding of sequencing in `IO`.

How does this impact us in practice? In addition to `catch`, another common type of operation is resource clean-up: often called *bracketing*. These are two examples of higher-order effects ubiquitous in real world software. It has been shown that `MonadBaseControl` fails to safely lift `bracket` through transformers [103]. Such erroneous behaviours cannot be allowed to pollute our applications. Helpfully, many of these concerns are shielded from users through opinionated libraries. For example, standard `IO` operations, including `concurrently`, are abstracted in terms of these type classes in *lifted-base* and *lifted-async*. The *unliftio* library goes one step further by defining a version of `MonadBaseControl` restricted to operations that are inherently *safe*. Given the design decisions we have deployed in the example of §6.3 this is particularly useful: `ReaderT`, with its read-only state, has standard interaction semantics with *all* types of lifted operations. Thus, we can safely lift useful `IO` combinators into the `App` interpreter.

In familiar fashion, we have techniques to workaround the challenging semantics of transformer interactions. Nevertheless, we should still count this as a point against transformers.

Repeating Effects. We still cannot duplicate class constraints in either functions or `deriving` clauses. This is inherent to Haskell's type class mechanism. In Scala, if a tagless final DSL varies in type parameters other than `F[_]` then it is possible to inject multiple instances of the trait to use sites. However, the need for this feature is significantly reduced given our new tools. The typical use case is multiple types of state: it is simple to achieve this by embedding references into the environment. Instead, since tagless final DSLs are cheap to define and noninvasive, we may prefer to push these duplicated effects behind an interface; giving better encapsulation in addition to facilitating multiple types of the same effect class.

There is still the theoretical argument that expressiveness should not be limited by the mechanics of our language. This is given more weight when considering that in practice custom effects are typically *isomorphic* to standard effects in their structure. Consider a `MonadSpaceStation` or a `MonadCache` wrapper around a caching system. These are comparable in shape to, respectively, `MonadReader` and `MonadState`. We might reasonably want to use a standard effect, with its pre-defined operations, as a vehicle for a custom effect: giving domain-specific semantics during interpretation. Extensible effects enable such an approach.

***B2**. Modularity* The example of §6.3 unequivocally improves upon §5.2 with respect to modularity. Previously, in order to propagate a given effect operation to the top of our interpreter it was necessary to change components of *other effects*: the n^2 instances problem. Using tagless final embedded DSLs this is no longer the case. Consequently, explicit lifting is significantly less appealing. Using `ReaderT`, `MonadReader` and `Has` to propagate global and local configuration and functions bolsters the credentials of the class constraint approach first introduced in §5.3.2.

One criticism is that the `MonadRef` constraint is stronger than it needs to be. Our program has the ability to create mutable references at will, even though we only need one. The alternative option is also unsatisfactory: we could embed an `MVar`—necessary because we do not have an initial state—in the environment. This forces us to muddy the business logic by handling this optionality. Instead, we might prefer to combine the best of both worlds by introducing a small custom interface to encapsulate this optional mutable reference:

```
class MonadSpaceStationState m where
  get :: m AppData
  put :: AppData -> m ()
```

Appendix A.6 contains a more complete example implementing this interface by combining `MVar`, `Has` and `MonadBase`.

B3. *Extensibility* We outline two compelling recipes for extending a program with a new effect based on the techniques introduced in §6.1 and §6.2, and applied in §6.3. The *dependency graph* discussion of §5.3.2 is re-examined in the context of these two design patterns.

The Tagless Final Extension Recipe. Our new formula is significantly shorter:

1. Write a class interface, now viewed as a *tagless final eDSL*: `MonadSpaceStation`
2. Write an implementation of the interface directly for either a concrete interpreter or an abstract monad *instantiated* to that interpreter.

These two steps are all that is required to start using new operations in effectful programs. The implementation of `MonadSpaceStation` can now be fulfilled by *any monad* satisfying the dependencies captured in its class constraints. This departs significantly from allowing only `SpaceStationT` to be the semantic vessel for the `fetchLocation` operation. The advantages of this adjustment are clear: we avoid the n^2 instances problem and explicit lifting for user-defined effects. Furthermore, our two steps represent a fulfilment of the expression problem: *1* constitutes an extension in the row dimension, a new DSL operation, and *2* embodies extension in the column dimension, a new interpreter. This is the most convincing solution we have seen and explains the popularity of the tagless final design pattern. There are some drawbacks primarily regarding usability; we delay these until the next section.

The Method Dictionary Extension Recipe. Using records of functions as our main tool for modular abstraction is only marginally more involved. Appendix A.7 transforms our program to represent `fetchLocation` using a function record instead of tagless final. In summary:

1. Declare an independent function (not a type class instance): `fetchLocationImpl`
2. Embed this function into the environment: `Env`
3. Write or auto-generate a `Has` instance to project the operation out of the environment.
4. Optional: write a convenience function to provide syntax for the operation.

This approach is arguably more self-consistent than the tagless final pattern, because both configuration *and functionality* are embedded in the environment. One downside is that extending a program with a new effect will break in the place we construct our environment. Thus, this does not solve the expression problem in the interpretation dimension. However, it is still significantly less invasive than the extension recipe of §5.3.2.

B4. *Ergonomics* It should already be clear from the analysis so far that this effect system has greatly improved ergonomics when compared with the transformer-heavy approach of §5.2. We follow a similar pattern to the discussion in §5.3.2, addressing the issues of boilerplate, the cognitive cost and the likelihood of abusing the framework by misunderstanding its subtleties.

Boilerplate. What are the sources of incidental complexity in our improved implementations? The dependence on `Has` for both configuration and capability passing produces several boilerplate instances. These are certainly more intuitive than the n^2 instances we have seen previously and, more importantly, they are *safer*. There are no sources of incomprehensible semantics which can lead to misuse. However, some may still consider these cumbersome. In practice, as Snoyman points out [93], the cost of this boilerplate becomes amortised in larger projects once the top-level environment and projection patterns have been established. Automation can also help: libraries provide utilities to generate these instances for us using template Haskell macros [104, 105].

More generally, the magic of meta-programming and automation can cause more damage to code maintainability than paying the manual boilerplate price. We have seen this in practice with the convoluted automatic type class derivation process described in §5.3.1. It can be especially intimidating for people new to a language. Therefore, we should avoid this in general application development unless the benefits are well-established and widely understood.

The tagless final approach to custom DSLs is essentially boilerplate-free: we define the abstraction and instantiation each in one place. In Scala, we may also inject it explicitly at the call-site. It is worth clarifying that the type annotations given to `trackSpaceStation` and `trackSpeed` are *optional*. If they are omitted then the compiler automatically infers `App` as the monad; unifying the different effect operations used. This is an important point, because discussions about the expression problem in the literature tend to make this assumption when evaluating whether *recompilation* is necessary [60, 106]. Thus, we will include type inference as part of our evaluation of ergonomics.

Type Inference. There are two concrete examples of type inference issues. Both relate to our use of an abstract monad and type classes; bolstering the argument for using concrete types. Firstly, it is possible for the dependencies of this custom effect to be satisfied in places other than the `App` interpreter. This is epitomised by the necessity of the type signature on `program`:

```
program :: (MonadAppCommon r m, Has Count r) => m ()
```

Without this ascription, the compiler attempts to resolve the use of `fetchLocation` in the body of `program` by looking for an instance of `MonadSpaceStation`. Several constraints are satisfied, but `MonadIO` is not so the compiler complains. Of course, we merely want to inherit the instance from the parent scope of `trackSpaceStation`; the compiler does not know this. Although not a serious problem, this is symptomatic of an underlying characteristic of final embeddings: they depart from the *building block* idea of the original literature on transformers [2, 63, 68]. Instead, we are, in a sense, *flattening* effectful DSLs by parameterising them directly by the interpreter. Importing a module can summon a type class instances unexpectedly. Consequently, we may be presented with confusing errors, or worse, changes to program behaviour by misunderstanding where our DSLs are instantiated.

In Haskell, avoiding orphan and overlapping instances mitigates these risks. In Scala it is common to declare methods to construct tagless final eDSL instances rather than allowing them to be available implicitly. The Scala example of Appendix B.2 does precisely this. We then explicitly call this constructor in a `given` clause at the injection site; typically close to the program entry point. This approach is helped by our dismissal of transformers.

Secondly, as soon as we introduce more than one instance of `Has` in the same context, ambiguity ensues. The compiler can usually select the correct instance based on the concrete types, but composing reader operations can necessitate an explicit type annotation as below:

```
sleepWithDelay :: (HasSleeper r m, Has Seconds r, MonadReader r m) => m ()
sleepWithDelay = readEnv >>= (sleep . secsToMicrosecs)
```

This is not a serious problem: it is often preferable to keep type annotations for readability. Of course, passing parameters to functions is an option and can be preferred in smaller scale programs. Maintaining and testing components which describe precisely the inputs they need without any redundancy will always improve the developer experience, regardless of how explicitly they are passed.

Cognitive Overhead. The extent to which programmers need to care about the mechanics of type class derivation has significantly reduced from §5.3.2. We do not need to spend any time thinking about custom transformer instances and standard transformers have theirs pre-defined.

In the functional Scala community, it has been argued that pervasive use of tagless final introduces a barrier to entry due to its complexity [96]. This argument carries more weight than in traditional functional languages like Haskell, Idris or Purescript: it is common for new Scala developers to have prior experience with primarily mainstream object-oriented languages such as Java, Kotlin or TypeScript. Therefore, concepts and language features such as monads, type classes, higher-kinded types, type lambdas and parametric polymorphism may be unfamiliar. Many of these are foundational to pure functional programming and therefore unavoidable. Moreover, one does not need a deep knowledge of these mechanisms to be able to apply the techniques we have outlined. For Java programmers, tagless final can be thought of as an interface, with interpretations being the concrete class implementations. Indeed, in Scala this is precisely what they compile down to. An understanding of the academic origins are unnecessary to apply these tools in practice.

Nevertheless, using concrete types is undoubtedly simpler since we are dealing with fewer abstractions. This simplicity tends to be the motivation behind opinionated effect systems such as `RIO` in Haskell and `ZIO` in Scala [75, 52]. Instead of polymorphism, they deploy localised environment restrictions to achieve modularity whilst using a concrete interpreter in effectful programs. As we have seen with method dictionaries, this regains some granularity. Polymorphism can be recovered when using a monomorphic effect monad. For example, when writing applications using the *rio* framework we can decide to specify functions in terms of MTL-style type class constraints [93]. This is achieved by *unlifting* functions defined in terms of the concrete `RIO` interpreter using `MonadUnliftIO`: essentially a safer but more restrictive version of `MonadBaseControl`. In addition to the lower barrier to entry, there will inevitably be less boilerplate. For instance, the pervasive `MonadReader` constraint is unnecessary since these operations are built into the effect system.

However, polymorphism is still appealing. We can be absolutely sure that no unexpected effects are introduced: "constraints liberate, liberties constrain" [107]. Type signatures are also highly communicative. In Scala 3, verbosity can be mitigated with techniques such as those described in §6.4.1. Type inference has also improved, though it is hard to compete with languages like Haskell. Furthermore, we have flexibility in our interpretation of programs written in terms of tagless final DSLs. They may be interpreted by less powerful concrete types or even *initial embeddings*.

This is particularly useful for testing. There are generally two things we want to do in unit tests: assert against the return value of a function—black-box testing—and assert that interactions with dependencies are as expected—white-box testing. Both of these can be achieved using an interpreter which combines the error and writer effects, consider:

```
type EitherWriter[L, A] = EitherT[[X] =>> Writer[L, X], Throwable, A]
```

This composes `Writer` (the writing part of `State`) with `Either` in a *state-preserving* manner; as we have discussed previously in §3.1.2 and §6.4.1. This transformer-based interpreter gives rise to an appealing pattern for testing, which we illustrate in Appendix B.4. We use writer to *log* interactions and `Either` to test failure. Since `Writer` has a `Monad` instance, `EitherT` has a `MonadThrow` instance—just like `IO`. Hence, it can instantiate our abstract `F[_]` in `trackSpaceStation` as long as we provide fake instances of our tagless final DSLs. We can avoid using an unnecessarily powerful concrete monad such as `cats.effect.IO`.

One final criticism is a more philosophical point than one with practical significance. We have partially regressed to a haphazard pattern comparable to the monolithic example of §3.3 or to ad-hoc effect syntax such as *async*/*await* in imperative languages. Although the framework undoubtedly has a vastly improved user experience to the transformer-heavy approach, as a whole it is less consistent. Furthermore, several of the new techniques rely on interpretation into the effect monad. It could be argued, therefore, that we are leaking implementation details into the syntax of our program. Programmers need to learn several orthogonal tools such as pure mutable references, exceptions, reader, tagless final. By their very nature, tagless final eDSLs are *coupled to context*. This is betrayed by DSL names such as `MonadRef` and their parameterisation over higher-order types. Monads are merely an implementation detail of the effect system interpretation and a tool for sequencing. The GADT approach described in §6.1 *is* context free; we contrast the two in the analysis of §7.4.

Susceptibility to Errors. Using tagless final, the ReaderT pattern and our recommendations, there are far fewer opportunities to make mistakes. Situations commanding the use of higher-order and control effects are more likely to cause concern. `MonadBaseControl` is a confusing construction: it is easy to misuse and produce unexpected program behaviour. Issues can be avoided by using library-defined operations and more opinionated frameworks such as *rio* and *unliftio* [93]. Moreover, we reiterate the point from §5.3.2: real world situations necessitating composition of higher-order and control operations are rare and inevitably call for caution. This same is true in Scala. Though the elimination of transformers as part of our general effect system makes the unlifting problem largely irrelevant, control effects inevitably have semantic challenges.

6.5 Compendium

We enumerate the key positives and negatives of the analysis, again using the criteria sets of **A1**–**A4** and **B1**–**B4** as a guide. In addition, negatives of §5.4 are referenced to highlight their resolution following our improvements. Some criticisms are more stubborn; symptomatic of the inherent limitations of transformers.

Positives

1. *Environment & Modularity*. The reader effect is a vehicle for configuration passing and dependency injection using *method dictionaries* and `Has`. This addresses **C1**.

2. *State, IO & Expressiveness*. We have a toolkit for mutating state in the presence of concurrency and guidelines for judicious usage; resolving **C2**. State transformers such as continue to be useful, such as `WriterT` for unit testing.

3. *Errors & IO*. For most applications imperative-style errors can be captured directly in the effect monad, removing the second error channel of **C3**. Transformers may still be preferred for localised explicit error tracking.

4. *Expressiveness*. **C4** is partially resolved: it is possible to express multiple instances of the same effect class by embedding state or functions in the environment.

5. *Modularity, Extensibility & Ergonomics*. We have broadly appeased the issues spawned from transformer composition: **C5**, **C6**, **C7**, **C9** and **C10**. This is a natural consequence of the shrinking of our stack size, mitigating the n^2 instance problem, lifting, and the general cognitive overhead associated with taming the transformer stack.

6. *Extensibility*. Tagless final eDSLs embody an appealing solution to the expression problem, solving **C8**. Method dictionaries provide another intuitive option.

Negatives

D1 *Environment & Ergonomics*. Using the ReaderT pattern, we must write boilerplate `Has` instances and syntax for effects. When using an abstract monad, we must additionally carry an extraneous `MonadReader` constraint throughout our business logic. Meta-programming can help but is not without cost.

D2 *Environment & Extensibility*. Embedding user-defined effects in method dictionaries demands making breaking—though isolated—changes to the environment.

D3 *Expressiveness*. The problem summarised by **C4** persists: perfectly reasonable programs are unrepresentable using transformers due to static ordering. Effect abstractions still cannot be repeated due to language-specific type class mechanics.

D4 *Expressiveness*. `lift` is not powerful enough for higher-order and control operations. We require more complex type classes which have challenging semantics.

D5 *Expressiveness & Ergonomics*. The necessity of several orthogonal techniques to express effects breeds inconsistency and betrays the theoretical limitations of transformers.

D6 *Ergonomics*. Tagless final eDSLs arguably have a barrier to entry in hybrid languages.

D7 *Ergonomics*. Type inference can suffer when using tagless final or the ReaderT pattern necessitating extraneous type annotations, which can be verbose.

This smaller list of grievances makes abundantly clear that we have improved upon the naive MTL implementation of §5.2. Our focus has generally shifted from modularity and extensibility to more inherent limitations in expressiveness and less serious issues of usability. These are better problems to have: the toolkit presented here is sufficient for the majority of real world applications. Functional programmers deploy many of these techniques in practice; explaining why MTL-style effect systems remain prevalent.

Nevertheless, it is always prudent to strive for better theoretical frameworks as well as making these kinds of adjustments based on practical conventional wisdom. An academic objection to our approach is that we are making pragmatic design decisions to work around fundamental limitations of transformers, and that we should instead focus on *wholesale improvements*. This would naturally lead to not only theoretical satisfaction, but typically also practical benefits. Indeed, extensible effect systems have been developed precisely with these goals in mind [22, 23, 108, 109]. The hypothetical ceiling appears to be higher, as we shall see in Chapter 7.

7 Extensible Effects

Chapters 5 and 6 revolved around monad transformers and MTL interfaces, along with recommendations to avoid common pitfalls and aid usability. With the context of §6.1, these interfaces can be viewed as *tagless final embedded DSLs*. To contrast, we introduced initial embeddings in the context of Hutton's razor [88], using sum types and GADTs. It is not yet clear how this approach would scale to larger problems and whether it is appropriate for general application development.

In fact, initial embeddings have formed the basis of fully-fledged programming patterns. Free monads and extensible effect systems embody two popular frameworks which emphasise *DSL-driven development* through the use of initial DSLs. It would be intellectually dishonest to compare these approaches with a naive implementation such as that presented in §5.2, so one of the goals of Chapter 6 was to facilitate a fair comparison.

The structure of this chapter broadly reflects the chronology of the constituent innovations. In §7.1 we begin by introducing a necessary prerequisite and common technique in functional programming: free monads. Our example is presented in §7.1.2 using Swiestra's initial formulation that combines coproducts and free monads [70]. This preceded breakthroughs by Kiselyov et al. [22, 23] and the formulation of *extensible effect systems*; which we cover in §7.2. After introducing these two tools, we reimplement the ISS example in §7.3 and perform a now familiar analysis, using the criteria sets of §2.2.1 and §4.1 to contrast extensible effects and free monads against transformers.

Though implementations exist in several languages such as OCaml, Scala, and Idris, we focus on Haskell which has garnered significant interest in the design space of extensible effect systems and is consistent with most of the examples. The issue of *performance* has been paid particular attention in both the academic and programming communities. In §7.4.3 we will have the necessary background to explore this topic in the context of *all* the effect systems we have covered. A final compendium in §7.5 captures our evaluation and constitutes a high-level summary of the effect system analyses throughout this document.

7.1 Free Monads

Despite monad transformers emerging as a workable solution to the problem of representing and composing effects in pure functional programs, many were unsatisfied with their limitations. Liang et al. and Wadler had been keen to emphasise that neither monads nor monad transformers were the panacea, but simply helpful techniques which formed the basis of useful programming patterns [1, 2].

Since early functional programming in ML, *initial DSL embeddings* (recall from §6.1) have been an appealing tool for expressing small domains without tying ourselves to context in the host language. However, given what we have seen so far, there are two obstacles to writing larger, real world applications:

E1. Building up complex expressions in terms of data types requires repetitive nesting of terms. Consider our example expressions from §6.1:

```
ex1 = Add (I 3) (I 4)
ex2 = Add ex1 (I 5)
```

One cannot foresee this being a feasible approach when scaled to industry programming, even with convenience functions for composing nested expressions.

E2. The expression problem statement [60]: initial DSL embeddings lack extensibility.

The first problem is avoided with a *final* embedding because we are able to use `do` notation for sequencing operations by virtue of DSLs being parameterised directly by a *monad*. The second problem we have covered extensively in §6.1: DSLs written using tagless final are extensible in both the instructions and the interpretations when represented as, respectively, type classes and type class instances. At its core, the issue is that sum types are *closed unions*. Any viable solution would need to contain some representation of an *open union*.

We reiterate the sentiment of §3.4 that the majority of software should be written in higher-level DSLs, with lower-level concerns being pushed to runtimes and libraries as much as possible [55, 56]. Furthermore, due to its lack of context, an initial DSL can be viewed as the most *primordial form* of a DSL; the closest we have to the *ultimate abstraction*, as Hudak puts it [54]. Therefore, an elegant solution to these problems would be an important tool for our toolkit.

7.1.1 Data Types à la Carte The idea of representing initial DSL embeddings as extensible unions was married with *free monads* into a coherent programming pattern by Swierstra in 2008 [70]. *Data types à la carte*, a succinct and celebrated paper, presents a recipe for creating composable DSLs using data types; solving both **E1** and **E2**. We describe the framework using the domain of our ISS program to illustrate the different elements. Consider modelling our DSL using purely data:

```
data Log f               = Log String f                   deriving Functor
data Sleep f             = Sleep Microseconds f           deriving Functor
newtype Ask r f          = Ask (r -> f)                   deriving Functor
newtype FetchLocation f = FetchLocation (Location -> f) deriving Functor
data Catch e f           = Catch f (e -> f)               deriving Functor
data State s f           = Get (s -> f) | Put s f         deriving Functor
```

The same names are used for consistency. Each independent data type embodies an instruction; together an instruction set. Why do we need the f? Since these definitions are not members of a union, in order to compose them we need to reintroduce recursion: somewhere to store the *rest of the program* since a program with a single instruction is not useful. With this construction we can write a small program:

```
-- prog :: Sleep (Log (Sleep Bool))
prog = Sleep (Microseconds 1) $ Log "Hi" $ Sleep (Microseconds 2) False
```

The performs a sleep, log, and sleep before returning a `Bool`. The inferred type is commented out. Calling these data constructors is tedious and verbose. As it stands, the type is coupled to the data—to interpret this program we would need to unwrap each layer manually. We can apply a trick borrowed from category theory to improve the usability:

```
data Free f a  = Pure a | Free (f (Free f a))
```

`Free` is a recursively defined sum type accepting a type constructor f and a type a. It is known as the *free monad*. We define convenience functions to construct `Free`-programs with the `State` DSL:

```
get :: Free (State s) s
get = inject $ Get pure
```

```
put :: s -> Free (State s) ()
put s = (inject . Put s) $ pure ()
```

The mathematical insight is that if f is a functor—has a lawful instance of `fmap` defined in Appendix D.1—then `Free f` is a monad. This explains the `Functor` derivation. Monads are appealing, because we can sequence them using do notation. For example:

```
increment :: Free (State Int) ()
increment = do
  i :: Int <- get
  put i + 1
```

We have pushed recursion to the free monad. The nesting of data is substituted for traditional monadic syntax, solving **E1**. This technique is useful on its own, since we now have an ergonomic way to sequence initial DSLs. However, there is still the issue of extensibility. These programs are written in terms of only a single instruction set, how can this be extended? With tagless final, an open union is achieved using type classes and instances. Instantiating an instance constitutes becoming a *member* of the open union for a particular DSL. We need a different mechanism for constructing an open union in which *data types* can reside.

Although not a novel innovation, Swierstra connected this DSL problem with another well-known mathematical property—that functors compose. One of the key insights, first proposed by Lüth and Ghani [110], was to take the *coproduct* of two functors to allow composition of free monads. If we intuit a sum type as a tagged union of types, then we can think of a coproduct as a *tagged union of type constructors*. We use the original idioms below, with the full framework in Appendix A.8:

```
data (:+:) f g a = InL (f a) | InR (g a) deriving Functor

class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a

instance Functor f => f :<: f where
  inj = id

instance (Functor f, Functor g) => f :<: (f :+: g) where
  inj = InL

instance {-# OVERLAPS #-} (Functor f, Functor g, Functor h, f :<: g) => f :<: (h :+: g) where
  inj = InR . inj

inject :: (f :<: g) => f (Free g a) -> Free g a
inject = Free . inj
```

`:+:` is similar to `Either`, except that its data constructors accept type constructors. The `:<:` class describes the *subtyping* relationship and is characterised by a function `inj` to *inject* a subtype into its supertype. The inductive instance definitions mean that we can read `f :<: g` as: the functor f occurs somewhere within the functor coproduct g. In our case this *supertype* is a potentially nested cascade of `:+:` types. Combining this with an injection into the `Free` monad,

we can *lift* programs parameterised by functors into programs parameterised by the functor coproduct. `OVERLAPS` can cause type inference problems due to ambiguous instances; this is discussed in §7.4.2. We improve our DSL syntax:

```
get :: (State s :<: f) => Free f s
get = inject $ Get pure

put :: (State s :<: f) => s -> Free f ()
put s = (inject . Put s) $ pure ()
```

Notably, the f is now an *abstract* open union and `State s` is a member. Our free monadic DSL is now *extensible*. Let's modify `increment` with logging:

```
-- increment :: (State Int :<: f, Log :<: f) => Free f ()
increment = do
  i :: Int <- get
  put i + 1
  log "Hello, World!"
```

The type signature is inferred as any coproduct f containing the operations `State Int` and `Log`. This powerful construction solves problem **E2** in the *row* dimension. We now have enough tools to implement the larger ISS example and discern whether *interpretation* of these instructions is comparably extensible.

7.1.2 Example Implementation: Free Monads The full example implemented using free monads is included in Appendix A.9. The bodies of our functions are—notably—almost identical to the naive example of §5.2. The most interesting point of discussion in the effectful program is the type signatures:

```
trackSpaceStation :: (Sleep :<: f, Log :<: f, FetchLocation :<: f,
                      Ask Config :<: f, Catch SomeException :<: f) => Free f ()

trackSpeed :: (Log :<: f, Sleep :<: f, FetchLocation :<: f, Ask Config :<: f)
                                          =>  Free (State AppData :+: f) ()
```

Our instructions are passed as *capabilities* to these functions using f as a vehicle. This will later be instantiated to a concrete coproduct containing the *full* instruction set. Our type signatures are expressive, communicating precisely the effects the function may perform; though it does not actually *perform* them. Recall that this is the *being* of effects, not the *doing*: we are sending these instructions somewhere else to get done. The exception is `State`. Much like `StateT` in §5.2, to satisfy these type signatures this effect must be introduced *and* eliminated within the body of `trackSpaceStation`.

```
execState :: Functor f => s -> Free (State s :+: f) a -> Free f s
execState s (Pure _)               = Pure s
execState s (Free (InL (Get k)))   = execState s (k s)
execState _ (Free (InL (Put s m))) = execState s m
execState s (Free (InR f))         = Free (execState s <$> f)
```

Figure 7.1: Free Monad State Interpreter

We define a function to interpret the `State` instruction just like `execStateT` interprets a program in `StateT`. This accepts a free-program consisting of the union of an arbitrary DSL—the functor f—with the state DSL, and eliminates the state component by threading `s` through recursive evaluations. This is the standard or *pure* state semantics, as opposed to *concurrent* semantics. `Free (InR f)` indicates that the current instruction is not one of `State`, so we simply continue evaluation of the remaining program f since it might contain further state instructions. We call this from the body of `trackSpaceStation`. Needing to explicitly project out of the coproduct with pattern matching lacks modularity since we must assume that `State` appears on a particular side. The evaluation technique employed by Swierstra embodies an elegant solution to this problem using type classes [70]:

```
class Functor f => Eval f m where
  eval :: f (m a) -> m a

instance (Eval f m, Eval g m) => Eval (f :+: g) m where
  eval (InL f) = eval f
  eval (InR g) = eval g
```

`Eval` encapsulates interpretation of each instruction. Given a coproduct with interpreters for the left and right functors, we can then trivially interpret the overall functor composition. Now implementations of instructions amount to instantiating this class for the corresponding data types:

```
instance (MonadReader Config m, MonadIO m) => Eval FetchLocation m where
  eval (FetchLocation useLocation) = do
    body <- MR.reader httpRequest >>= httpJSON
    getResponseBody body `tapM` doLog >>= useLocation

instance MonadIO m => Eval Sleep m where
  eval (Sleep ms a) = liftIO (threadDelay $ getMicroseconds ms) >> a
```

The target of this interpretation is an abstract monad which will eventually be ripped down to `IO`, as before. We see that the familiar MTL-style class abstractions are still useful for providing *semantics* to our instructions. It is now transparent that the f in our DSL definitions represents the rest of the program: we sequence it after the evaluation of this particular instruction using monadic combinators `>>=` and `>>`. The final piece is the concrete interpreter:

```
type Effects r e = Ask r :+: (Sleep :+: (Log :+: (Catch e :+: FetchLocation)))
newtype App a    = App { runApp :: Config -> IO a } deriving (...)

main = let program = trackSpaceStation :: Free (Effects Config SomeException) ()
       in runApp (iterM eval program) config
```

We omit the long list of class derivations. Save the environment type, this `App` is *identical* to our ReaderT-pattern interpreter of §6.3. To tear down the free monadic structure, thereby eliminating the DSL in favour of some target monad, we feed `eval` to the standard function `iterM`. This *iterates* through the layers of `Free`, peeling off instructions by feeding them to the corresponding `eval` implementation. As Kiselyov notes, this *folding* over a free monadic program parameterised by a functor—a generalisation of folding over lists—was one of the key contributions of Swierstra's work [106]. We include a more detailed explanation of the steps executed by `iterM` in Appendix C.4.

There are some immediate criticisms to make. Needing to explicitly specify the order of instructions within the DSL coproduct is unfortunate. As is having to provide a type ascription to nudge the compiler in its inductive resolution of the `Eval` instance. Note that, although the relative order of the class constraints in `trackSpaceStation` is immaterial, the brackets in the interpretation type *are* necessary for this instance resolution. Type inference is a common problem in this framework, due partially to the overlapping instance for `:<:`. All of this is incidental complexity.

A number of our design decisions are somewhat contrived to illustrate coproducts and free monads in a manner most directly comparable to the naive transformer example of §5.2. There are several improvements we could make based on what we have learnt in Chapter 6. The `Ask` instruction is leaking implementation details from the reader effect; domain-specific instructions such as `GetDelay` would be more appropriate. We can also make use of the `Has` pattern in the *interpretations* of these instructions, for more granular injection of configuration and functions. An improved implementation, combining elements of MTL, the ReaderT pattern and this new approach is included in Appendix A.10.

Holistically, the solution is appealing. It has the particular benefit of a clear separation between syntax and semantics; the *being* and the *doing* of effects. This technique has had phases of popularity within different functional programming communities and remains a useful tool. It is important to emphasise that free monads and coproducts are separate techniques and can be applied independently. In some situations extensibility is not a concern, so a regular (G)ADT is an adequate model of the domain. We delineate between these patterns in more depth in §7.4.2: our final assessment of extensibility.

Free monads and coproducts formed the basis of the early extensible effect systems. However, owing to Kiselyov et al. [23] and frameworks developed by the Haskell community, extensible effects have improved upon and arguably subsumed the approach described here. Therefore, rather than evaluating it in isolation, we will interleave commentary throughout the §7.4 analysis of extensible effects. Hereafter, the *Data types à la carte* paper and its framework [70]—specifically, the combination of free monads, coproducts and evaluation type classes—will be abbreviated as *DTalC*.

7.2 Origins

Chapter 6 attempted to show how best practices can emerge as a result of unifying academic innovations with practical software engineering problems. The story of extensible effects follows a similar pattern. We begin with their emergence in the context of programming language research before presenting several production-worthy libraries spawned from the literature.

We can trace the initial ideas back to Cartwright and Felleisen in 1994 [111]. The key insight was conceptualising effects as interactions between syntactical language constructs and a central *authority*. Thus, effects can be viewed as *requests*. The authority acts like an operating system, controlling lower-level resources. In deciding how to respond to requests, it embodies the *doing* of effects; enriching the operations with semantics. This paper was referenced in the seminal work of Liang et al. on monad transformers and treated with caution since the connection to their approach was not yet clear [2]. With hindsight, we illuminate this overlap in the contexts of both algebraic and extensible effects.

Algebraic effect and handlers, attributable to Plotkin and Pretnar [6, 12], spawned from this early work. *Algebraic effect systems* can be characterised using the intuition of *resumable exception handlers*: generalising imperative language constructions such as *throw*/*try*/*catch* and *async*/*await* [71, 112]. This has generated a flurry of activity in programming language research. Several languages have been developed with first-class support for algebraic effects and handlers [13, 14, 15, 16, 17, 18]. Library embeddings in more production-ready languages such as Scala and Typescript have generated an increased awareness of algebraic effects within more mainstream software engineering communities [19, 20, 21, 113, 114]. Though we have already considered a detailed exploration of this domain out of scope, it is highly congruent to extensible effects and constitutes a promising avenue for future work.

Extensible Effects. Independently, Kiselyov et al. explored the connection between the early work of Cartwright and Felleisen [111], and *pure functional* effect systems in languages such as Haskell [22]. Inspired by these foundational ideas, coproducts [110], free monads [70], and the popular contemporary MTL programming style, the *extensible effects* framework was formulated. To give an intuition, comparisons can be drawn with the effect systems we have covered. The effect *requests* are analogous to MTL type class operations or free monadic data constructors. The execution of effects by the central *authority* is conceptually comparable to concrete transformers or interpreters of free monadic structures such as `Eval` and `execState` of §7.1.2. The authority then resumes the program by calling a continuation, akin to the monadic `>>=` operator and the type parameter f in initially embedded instructions.

The limitations of the *central authority* framework were highlighted: extensibility, modularity and type-safety. These problems were addressed by instead constructing a *distributed bureaucracy* with simple functions for both sending and receiving effect "requests". The improved framework consisted of an open union from which individual effects could be interpreted using *handlers*. This open union is represented as a type-indexed coproduct of functors; a generalisation of the free monadic composition popularised by *DTalC* [70]. Intuitively, this is a list of type constructors expressing effects, for example: `Reader r`, `Either e` and `State s`. This formed the basis of a novel type-and-effect system with an emphasis on the benefits over monad transformers. Two notable advantages are that effect ordering can be ignored when irrelevant, and we can specify different compositional semantics within a single program—the §7.4.2 analysis of expressiveness examines this in more depth. A Haskell library was used demonstrate the practical ergonomics of this approach [115].

The whole framework is designed with extensibility in mind; it is simple to include our own DSLs in this open union. Hence the name, *extensible effects*. It is further shown that MTL-style classes are subsumed by extensible effects by expressing them in terms of the extensible `Eff` monad. This framework allows us to write initial DSL embeddings as pure data, *send* them as instructions to the effect system, and eliminate them using handler functions. We have a compelling tool for DSL-driven development. However, there were two sources of boilerplate: the need to derive `Functor` for every instruction, exemplified by §7.1.1, and an extraneous `Typeable` constraint necessary in operations as evidence for effect membership in the open union. These detracted from usability and constituted most of the early criticisms of extensible effects. Moreover, free monads classically suffer from performance issues: we are repetitively constructing and deconstructing trees of data in memory.

Freer Monads. The discovery of the so-called *free-er* monad remedied these unsatisfactory properties [23]. This construction liberated free monads from the functor constraint by embedding its characteristic `fmap` function *within* a GADT representation of the free monad data type. Extension could be achieved by composing effects using a modification of the coproduct open union described in [22]. This new representation differs significantly from that of the classic free monad in §7.1.1. The freer monad, with this `Union` embedded within it, is defined as follows:

```
data FEFree r a where
  Pure   :: a -> FEFree r a
  Impure :: Union r x -> (x -> FEFree r a) -> FEFree r a
```

The resulting framework was a breakthrough. The most practically significant consequence of this innovation was that users are protected from effect system implementation details such functors and coproducts. Instead, we need only write independent instructions as GADTs and syntax to *send* them into the effect system; we shall see this next in §7.3. The problem of performance was addressed by applying *reductions* to nested freer structures; an opportunity enabled by the embedding of `fmap` in an *initial* form rather than its classical *final* representation in the `Functor` type class.

Extensible Effects for Application Development. Following this initial work, a series of extensible effect frameworks have been released over the last few years, typically with the goal of providing a viable alternative to MTL. These all use the freer monad construction to some degree, with differences stemming from different focusses. Each improves upon the previous in a particular dimension: usually either expressiveness, ergonomics or speed.

A popular early framework *freer-simple* aimed to provide a clean API for common use cases, at the expense of support for custom higher-order effects (HOEs) [116]. The work of Wu et al. in the context of algebraic effects provide greater flexibility in the representations of HOEs in extensible effect systems [117]. Moreover, the *fusion* technique described by Wu and Schrijvers for improving the efficiency of effect handlers was first implemented in the *fused-effects* framework [61, 109]. More recently, inspired by this research, *polysemy* and *eff* were designed with the goal of maximal ergonomics without sacrificing performance [108, 118]. The former has become popular in industrial Haskell programming and the author is a vocal advocate of using extensible effect systems in production; providing several useful resources [119]. For this reason, we use *polysemy* as the vehicle for demonstrating extensible effects in the next section. *eff* is uncompromising in its approach and is not yet released due to performance optimisations built upon changes to the Haskell compiler [62]. It is still a promising effect system for future adoption. *Most* recently, *in-other-words* and *effectful* have been developed with particular focusses on, respectively, simplified HOEs and speed. This is an open design space; we are likely to see further innovations.

Unsurprisingly, Scala has lagged behind in terms of adoption. There are library embeddings of extensible and algebraic effects, though these are not widely known with minimal industry adoption [120, 121]. Improvements in

Scala 3, such as the introduction of union types and polymorphic function values, encourage novel encodings of freer monads. More frameworks are expected to emerge in this domain, along with a gradual recognition of the merits of extensible effects as a design pattern.

Throughout the work of Kiselyov et al. [22, 23], there is an emphasis on the advantages of extensible effects over monad transformers. Improvements in expressiveness (**C4**, **D3**) and extensibility (**C5**, **C6**, **C7**, **C8**, **D2**) are undisputable. A more interesting quandary, and a key goal of this thesis project, is the usability of extensible effect systems holistically. This shall be central to the analysis of §7.4, though for completeness we must evaluate these claims against *all* of our criteria.

7.3 Example Implementation: Extensible Effects

The full code for this section is in Appendix A.11. We begin with our custom I/O effects:

```
data Sleep m a          where Sleep :: Microseconds -> Sleep m ()
data FetchLocation m a where FetchLocation :: FetchLocation m Location
```

Like the example of §7.1.1, this is an initial embedding. There are two obvious differences: it uses GADTs, introduced in §6.1, and there is no `Functor` derivation thanks to the freer monad insight [23]. The *m* parameter reintroduces recursion; representing the rest of the program. Again we write convenient syntax for constructing composable expressions involving these data types:

```
sleep :: Member Sleep r => Microseconds -> Sem r ()
sleep = P.send . Sleep

fetchLocation :: Member FetchLocation r => Sem r Location
fetchLocation = P.send FetchLocation
```

At the centre of this implementation is the `Sem` effect type, native to the *polysemy* library [108]. The type constructor `r` represents a coproduct of functors, using `Member` to express *membership* in this open union. This gives rise to notably concise syntax. We use `P.send` to *send* these effects to the distributed bureaucracy described by Kiselyov et al. [22]. This contrasts with the use of concrete transformers in the naive example of §5.2, in which the *return type* tells us which effects a function might perform. As is customary, these repetitive functions can be automatically generated through meta-programming: most extensible effect systems provide this facility and polysemy *recommends* it. We can intuit that these functions *raise* effects into the effect system—as purely *syntactic* constructs—to be later given semantics by *handlers*. Composing two effects is barely more verbose:

```
sleepWithDelay :: Members '[Sleep, Reader Config] r => Sem r ()
sleepWithDelay = asks requestDelay >>= (sleep . secsToMicrosecs)
```

Our GADTs give rise to functors via the freer construction; these can then be combined using coproducts. The result is a terse *type-level list* of effects, captured by `Members` and `'[]`. As with the free monad example, we omit the function bodies which are almost identical. Combining these user-defined capabilities with standard effects give us informative function signatures:

```
trackSpaceStation :: Members '[Sleep, Trace, FetchLocation, Reader Config, Error ISSError] r => Sem r ()
trackSpeed        :: Members '[Sleep, Trace, FetchLocation, Reader Config, State AppData] r => Sem r ()
```

Standard effects are provided by *polysemy*: `Reader`, `Error`, `State` and `Trace` for logging. Interpretation is now represented using functions as opposed to type class instances:

```
runFetchLocation :: Members '[Embed IO, Reader Config, Trace, Error ISSError] r
                    => Sem (FetchLocation ': r) a -> Sem r a
runFetchLocation = interpret \case
  FetchLocation -> do
    request <- asks httpRequest
    body    <- fromExceptionVia ISSError (liftIO $ httpJSON request)
    getResponseBody body `tapM` log

sleepToIO :: Member (Embed IO) r => Sem (Sleep ': r) a -> Sem r a
sleepToIO = interpret \case Sleep ms -> embed (threadDelay $ getMicroseconds ms)
```

To gain an intuition we examine the simple `Sleep` effect. The input to `sleepToIO` is a program written in terms of `Sem`. In the same way one would take the head of a linked list using `x:xs`, the `':` takes the head of the *type-level* list of effects. The member `Embed IO` is comparable to `MonadIO` of previous examples, allowing us to `embed` programs in `IO` into `Sem`. Conceptually, we are *trading* our user-defined `Sleep` for a more powerful effect. The same is happening in `runFetchLocation`, except that there are several effects involved in interpretation of `FetchLocation`. Importantly, the order is not significant until we take the head off the list with `':`. Thus, the union has set-like semantics, so the interpreter can be run against *any* program containing `Sleep`. Other effects in the coproduct `r` are *opaque* to this

function. This is comparable to transformer interpreters such as `runReaderT`: the underlying monad—which itself might be a monad transformer—is abstracted away. All that remains is the top-level interpretation:

```
interpretISS :: Sem '[FetchLocation, Error ISSError, Sleep, Log, Reader Config,
                      Embed IO] a -> IO ()
interpretISS prog = runFetchLocation prog & runError @ISSError & sleepToIO &
                            traceToStdout & runReader config & runM & void
```

Figure 7.2: Extensible Effects Final Interpreter

Effect elimination is achieved by removing effects one at a time from the coproduct list. We are exiting the *polysemy* effect system and delivering a raw `IO` to the runtime with `runM`. This is directly comparable to the transformer interpretation function of Figure 5.1. Instead of a *vertical transformer stack*, we have a *horizontal list* of effects. Predictably, this has implications for compositional semantics; discussed in §7.4.2.

7.4 Analysis

We will assess the two implementations based on initial DSL embeddings together: free monads of §7.1.2 and extensible effects of §7.3. Our analysis will be punctuated by references to the compendium negatives **C1**–**C10** and **D1**–**D7**, with the objective of mitigating or outright solving as many as possible. As always, variations of these techniques will be discussed along with the relevant trade-offs.

7.4.1 Representing Effects

A1. *Environment* In contrast to the implementation of §6.3, we are back to using `Reader` only for configuration passing. Dependency injection is achieved through membership in the union of effects; there is no need for `Has` and the corresponding boilerplate instances. However, the problem of **C1** has resurfaced: the concrete environment `Config` is nonmodular. How might we regain modularity using extensible effects?

The choice to solve this problem using an abstract environment and `Has` was forced upon us by the inability to duplicate the `MonadReader` effect with different environment types—a product of functional dependencies in the MTL class definitions. We use this as an opportunity to demonstrate the utility of *repeating* effects: the **D3** criticism. Since the open union is *type-indexed*, as long as one of the type parameters varies in a particular member, we can duplicate that member. For example:

```
trackSpaceStation :: Members '[Reader Count, Reader Seconds, ...] r => ...
runFetchLocation  :: Members '[Reader Request, ...] r => ...
```

We make the environments more fine-grained both in the effectful program and the `FetchLocation` handler. The compiler prevents us from forgetting to handle an effect; if the program passed to `interpretISS` contains an effect outside the union, we get a useful error:

```
  • Unhandled effect 'Reader Seconds'
    Probable fix:
      add an interpretation for 'Reader Seconds'
```

We can satisfy the compiler by writing a new interpreter in terms of the standard `runReader`:

```
runReader3 :: i -> (i -> j) -> (i -> k) -> (i -> l) ->
              Sem (Reader j ': Reader k ': Reader l ': r) a -> Sem r a
runReader3 i f g h prog = runReader (h i) $ runReader (g i) $ runReader (f i) prog
```

This projects three components of the environment, eliminating, in order, each effect from the list. This may be abstracted into a *fold* over an arbitrary number of projections. We call this interpreter in the body of `interpretISS`. Appendix A.12 contains a full example incorporating this improvement, along with further modifications proposed later in the analysis.

We may consider the duplication too verbose and prefer to just use `Config`. In this case, we can write another interpreter in terms of the standard interpreter which projects a particular element of the environment at the *call-site* to a function.

```
runProjectedReader :: Member (Reader i) r => (i -> j) -> Sem (Reader j ': r) a -> Sem r a
runProjectedReader f prog = asks f >>= (`runReader` prog)
```

Note that this is not unique to extensible effects; one could write a `runProjectedReaderT` achieving the same goal. However, it cannot be *duplicated* without using two stacked concrete transformers. To illustrate, consider a large environment `BigConfig`:

```
data BigConfig = BigConfig { _foo :: Foo, _bar :: Bar }
```

We assume `_foo` and `_bar` are deeply nested and unrelated to each other. Using MTL, if we want to pass only these two elements to a function, we must group them together in, say, `FooBar`. Then the function would accept a `MonadReader m FooBar` constraint. Using extensible effects this is trivial:

```
callUseFooBar :: Member (Reader BigConfig) r => Sem r ()
callUseFooBar = runProjectedReader _foo $ runProjectedReader _bar useFooBar

useFooBar :: Members '[Reader Foo, Reader Bar] r => Sem r ()
```

We can duplicate `Reader` and call `runProjectedReader` twice, with the type being inferred based on the corresponding record accessor. Since we know that order is irrelevant for reader effects, this is an undisputable improvement upon §6.3. Three effect types are reduced to two, there is no `Has` boilerplate, and type inference is more consistent. Thus, our solution to **D3** additionally solves **D1** and partially **D7**. We see that the ability to express multiple instances of the same effect class gives us more flexibility while retaining the modularity benefits of a polymorphic environment.

A2. *State* The state effect is again used to demonstrate *partial evaluation*—interpreting an effect deeper within a program. Here this means eliminating `State` from our set of effects with `execState`. This is comparable to the use of `execStateT` in §5.2. However, previously we were forced to make a reluctant choice between a concrete transformer and explicit lifting or an abstract monad, `MonadState` and boilerplate instances to propagate custom effects into `StateT`. Instead, we now use an abstract union without the downsides. This is a definite reduction in incidental complexity and improves usability.

Concurrent State. Previously we addressed **C2** by using mutable references, perhaps encapsulated behind modular abstractions. In contrast, polysemy provides an `AtomicState` GADT with accompanying atomic state operations. Immediately we can include this effect to gain access to operations:

```
trackSpeed :: Members '[Sleep, ..., AtomicState AppData] r => Sem r ()
```

How do we eliminate it? Conceptually, concurrent state operations are strictly more powerful than regular state operations. That is, `State` should be a *specialisation* of `AtomicState` isolated to a single-threaded environment. Recall from §3.1.2 that traditional state interpreters carry the state value in a pair alongside the monadic value: this is how `StateT`, the free monadic `execState` of §7.1.2 and the regular `execState` here are implemented. Happily, polysemy provides an interpreter that eliminates `AtomicState` by *deferring* to the handler for `State`, shown in Figure 7.3. This

```
execAtomicStateViaState :: s -> Sem (AtomicState s ': r) a -> Sem r s
```

Figure 7.3: *Pure* Effect Handler for Atomic State

is useful if we want to support concurrent state, but not *impose it* on callers. For instance, in unit tests we may want to test our program in the absence of nondeterminism using the regular state semantics. This so-called *pure* effect handler facilitates such an endeavour—it does not interpret the effect in terms of other effects. The implementation in Appendix A.12 includes this change: `trackSpeed` emits an `AtomicState` and we employ the proxy interpreter above. If, instead, we wanted to make concurrent calls to the ISS API in production code, then we can provide true concurrent semantics with:

```
atomicStateToIO :: Member (Embed IO) r => s -> Sem (AtomicState s ': r) a -> Sem r (s, a)
```

This is a common situation in industry software where most applications are I/O bound—time is generally spent awaiting the results of asynchronous operations like network requests. At its core, the implementation uses the same effectful primitives that we described in §6.4.1 and used in the implementation of §6.3 via the `MonadRef` abstraction. `Embed IO` indicates that we are trading `AtomicState` for `IO` operations that are abstracted from us. We could take a similar approach to §6.4.2, further abstracting the use of `IO` behind our own DSL operations.

That the framework gives us a native primitive for concurrent state is a clear improvement upon `StateT`. It is also less ad-hoc than the use `IORef` and `MVar` throughout Chapter 6. In the previous section we illustrated how multiple `Reader` effects could be used in the same program. It would be similarly simple to interleave multiple instances of `State` and `AtomicState` throughout a program—perhaps with different semantics in different places, addressing **C4** and **D3**.

A3. *Errors* We have now gone one step further than just typed errors. Although errors thrown by `httpJSON` are *embedded* in `ISSError`, this data type is not *itself* an exception. This is a modularity improvement: exceptions are an implementation detail of Haskell's runtime, captured in the `IO` monad. It constitutes a shift away from the frequent exposure to these lower-level details in §6.3, instead hiding them behind higher-level abstractions. How exactly is this achieved? Errors are embedded into the extensible effect system using a convenience function `fromExceptionVia` provided by polysemy:

```
fromExceptionVia :: (Exception exc, Member (Error err) r, Member (Embed IO) r)
                 => (exc -> err) -> IO a -> Sem r a
```

It accepts two parameters: a function to convert exceptions raised in `IO` to user-defined errors, and the program which might throw an exception. The `Embed IO` constraint indicates that we need the ability to convert `IO`-programs to `Sem`-programs, and `Error err` ensures that the custom error effect is in our union. The result is a powerful mechanism for trading low-level error representations for high-level, custom errors encapsulated by the extensible effect monad. One step further would be to completely abstract exceptions from users in general application development. Thus, building a library ecosystem around extensible effect systems could lead to an improved developer experience.

Error Effect Elimination. One point of contention is that in the final interpretation of Figure 7.2 we are forced to discard the `Either` produced by `runError` with `void`. This gives a value of type `Sem r ()`. It was also necessary in the naive transformer implementation of §5.2. Though not a serious issue, this is symptomatic of a lack of expressiveness. Despite no errors being thrown following the `catch` in `trackSpaceStation`, this is not captured statically in our type signatures. Java checked exceptions and algebraic effects are examples of effect systems that perform this *elimination* of error effects upon handling a particular type of error. Replicating this behaviour would constitute an improvement to the §7.3 implementation.

The problem stems from failing to interpret errors at the single instance of `catch`. Theoretically, the program could throw further errors, for example in the handler function `logFailure`. Clearly, something is amiss. It would help to scrutinise exactly what we do in the cases of success and failure. In both cases, we write a log line. Abstractly, we are *trading* an `Error` effect for a `Log` effect. This is a familiar pattern: all of our interpretation functions exhibit this behaviour. Let's try to encapsulate this idea concretely:

```
errorToLog :: Member Log r => Sem (Error ISSError ': r) Speeds -> Sem r ()
errorToLog prog = runError prog >>= \case
  Left (ISSError e) -> log $ "Failed to call ISS with error: " ++ show e
  Right finalSpeeds ->
    log $ "Successfully tracked ISS speed, result is: " ++ show finalSpeeds
```

Given that the set of effects contains `Log`, and the program returns a `Speeds` value, this handler removes `Error` from the set. This is precisely the semantics we wanted, and a more idiomatic employment of extensible effects—`trackSpaceStation` no longer performs `catch`, leaving the responsibility of error handling to callers. We use this interpreter in the improved implementation of Appendix A.12. Our sequential running of handlers is now more expressive:

```
runFetchLocation prog & errorToLog & sleepToIO & logToIO &
  runReader3 config totalRequests httpRequest requestDelay & runM
```

`void` in no longer necessary, since `errorToLog` eliminates the `Either`. Moreover, this better captures our semantics. The order of handler application determines the order of removal of effects from the set. It is now significant that `logToIO` is applied *after* `errorToLog`, since we cannot use the `Log` effect after its removal. Without looking at any of the type signatures, the naming convention for handlers helps communicate that `runFetchLocation` uses `Error`, `Sleep`, `Log`, `Reader`, and perhaps some embedding of `IO` due to `runM`. The result is a more expressive and modular interpretation.

We should clarify that this ability is not unique to extensible effects. It is possible to eliminate the error component of the transformer stack in §5.2, but it would not be easy. We would need to rearrange the order of the stack to allow us to peel off the `ExceptT` layer first. Conversely, this was not necessary in the examples of §6.3 and §7.1.2 because errors are pushed to the base effect monad. Thus, throwing and catching exceptions are interpreted into the same type: `IO`.

This exercise demonstrates the power of the modular interpreter functions provided by extensible effect systems. Refactorings such as these may be more difficult to spot in MTL-style applications, due to the more cumbersome extension process and less obvious separation of syntax and semantics.

***A4**. I/O* In the spirit of DSL-driven development, we began by defining custom I/O effects as pure data without any context—one of the characteristics of initial embeddings. Given syntax for injection, in both the *DTalC* and extensible effect implementations we can immediately embed these effects into programs described in terms of open unions (coproducts). Semantics are postponed until later.

Trading Effects. Throughout the evaluations we have described the idea of trading custom I/O effects for other, lower-level I/O effects. This is a useful means of assessing the expressive power of each approach. How does this compare with the other effect systems we have covered? To illustrate we use `FetchLocation` and `Log`—every implementation of `fetchLocation` was dependent on the `log` operation as well as those of the HTTP library. The method of injection, however, varied significantly.

In the naive example of §5.2 we used a `MonadLogger` constraint, instantiated to `LoggingT` in the `App` transformer stack. Using method dictionaries in §6.3, we employed `MonadReader` and the `HasLogger r m` projection to log based on functions embedded in the environment. The *DTalC* example of §7.1.2 used `MonadIO` directly, which is symptomatic of a larger problem. We are defining operations *twice*: once in the initial DSL embedding with `log`, and once in

the interpretation with `doLog`. In essence, *instructions cannot be used in the interpreters of other instructions*. This redundancy betrays an inherent lack of modularity with the framework which will be discussed in §7.4.2.

In terms of the usability, this mix of two design patterns breeds inconsistency—initial DSL embeddings with coproducts for syntax, and tagless final classes for semantics. This commands an understanding of most of the tools described in both Chapters 6 and 7. In order to write programs in terms of initial DSL embeddings as much as possible, advocates for techniques such as *hierarchical free monads* propose structuring applications with different DSL *layers* operating at different levels of abstraction [122]. This is consistent with our preference for pushing lower-level concerns to the edges. In our case, we could have **`FetchLocation`** at a layer above **`Log`**, so that the former can be interpreted in terms of the latter. The original proposal encourages basic (G)ADTs to avoid coproduct boilerplate, accepting the ensuing lack of extensibility (covered in 6.1) as a worthwhile trade-off.

However, the more intrinsic problem persists. Eventually we must interpret the bottom-most DSL in terms of **`IO`**, and, when we do, we will inevitably be exposed to runtime details. To improve modularity, we may reach for MTL-style tools such as **`ReaderT`** and **`Has`**—we are back at inconsistency. Extensible effect systems satisfy these properties in addition to the benefits associated with writing programs in terms of *abstract* and *open* DSL unions.

For these reasons, we claim that extensible effects broadly subsume the *DTalC* pattern as a general effect system. In terms of consistency, both the ReaderT pattern and extensible effects are more appealing than an ad-hoc approaches built on free monads, exemplified by our §7.1.2 implementation. If, for instance, a large application is already built upon transformers and one wants to avoid committing to a whole new effect system, then free monads can still be a useful tool for DSL-driven development in smaller, more specific domains and subsets of applications.

Effect Isomorphism. In §5.3.2 we argued that the number of user-defined effects tends to dwarf standard effects in software applications. Moreover, these custom operations can be broadly categorised as standard operations based on their structure. Consider these two operations:

```
tell :: Member (Writer o) r => o -> Sem r ()
log :: (Member Log r, Show a)  => a -> Sem r ()
```

Recall from §4.2.2 and §6.4.1 that **`Writer`** effect is a way to *log* values with cumulative semantics. The `tell` operation is structurally identical to our `log`: is it possible to defer to **`Writer`**? This would avoid explicit interactions with the extensible effect system *authority* using **`P`**`.send` and `interpret` for, respectively, introducing and eliminating effects. Instead, we can forward *requests* to a writer operation and handler with three changes:

```
type Log      = Writer [String]
log a         = tell [show a]
logToIO prog = runWriter prog >>= (\(logs, v) -> traverse (embed . putStrLn) logs $> v)
```

This is notably noninvasive—simply a type alias instead of a new GADT. The type signatures are identical, except that we introduce a log effect with `tell` and eliminate it with `runWriter`. This represents a change in semantics: logs here are accumulated before writing to stdout at the end of the program. Many applications consist of predominantly user-defined effects which are structurally equivalent to **`Reader`**, **`Writer`**, and **`State`**. Suffice to say that writing custom effects in terms of standard transformers is not this simple. The challenging semantics of **`StateT`** and **`WriterT`** reduce the appeal, and the inability to duplicate effects limits the situations in which one can apply this trick. Recognising such patterns has compounding benefits within large repetitive codebases, which are pervasive in industry settings. Therefore, we consider this flexibility a point in favour of extensible effect systems.

7.4.2 Composing Effects

B1. *Expressiveness* After improving upon the naive implementation of §5.2 in §6.3, we were left with a number of more fundamental limitations of transformers. We use the remaining negatives of §6.5 as a guide; evaluating whether the implementations based on initial embeddings, §7.1.2 and §7.3, resolve these issues.

C4 *&* ***D3***. *Inflexible Compositional Semantics.* In §5.3.2 we introduced two effect compositions which are challenging to express using transformers, credited to Kiselyov et al. [22]. Composing errors and nondeterminism required stacking error transformers—equivalent to **`ExceptT`**—either side of a **`ListT`**. A second situation, demanding the composition of continuations and thread-local environment, exposed that transformers are wholly insufficient for expressing certain problem domains. The design decisions of Chapter 6 proved adequate to express our ISS example in a satisfactory manner. This was achieved by essentially pushing errors and state into the **`IO`** monad, and making extensive use of the *safe* **`ReaderT`** transformer. Although this gives rise to a more disjointed programming pattern, drawing on a number of distinct functional programming constructions, it is sufficient for the majority of real world situations.

It could be argued, however, that these design choices merely cover up the inherent limitations of transformers. We previously dismissed the more subtle situations as seldom practically significant. Though this may be true, a framework that can satisfy both vanilla effectful programs and more complex edge cases consistently has a high theoretical appeal. Extensible effect systems embody such a framework. Kiselyov et al. expressed those difficult compositions using the original *extensible-effects* library with relative ease [115]. Following advancements such as the freeing of the free

monad from the functor constraint [23] and the development of more ergonomic libraries such as polysemy, these challenging situations can be expressed with even greater succinctness. The solutions to these problems have broader implications than just these theoretical edge cases. For example, allowing effect repetition gives rise to appealing patterns for general application development—the `Has` pattern of §6.2 is subsumed by the repetition of `Reader` in §7.4.1.

What about the *DTalC* approach? Since our DSL is *context-free*, we can express whatever we like, including these difficult situations, in the program *syntax*. However, as soon as we add context, that is, interpret our instructions, we are at the mercy of the interpreter. In the §7.1.2, this is essentially `ReaderT` over `IO`. Hence, we will stumble upon the same limitations inherent to transformers.

Composing State and Errors. We examine the compositional semantics of state and errors for every effect system we have covered, following up on Sections 3.1.2, 5.3.2 and 6.4.1. We take inspiration from a so-called *semantic zoo* presented by King [48]. The examples are categorised as follows:

1. Raw `IO`: applying the recommendations of the ReaderT pattern in §6.2 we push both errors and (concurrent) mutable state into the `IO` monad.
2. MTL-style programming: as in the example of §5.2 we deploy the `ExceptT` and `StateT` transformers. Transformer ordering determines the semantics.
3. Extensible effects: we use `AtomicState` to illustrate both *pure* and concurrent semantics when handling the same effect. The handler ordering and the handler itself determine semantics.

In all cases we use the same toy function, translated to the appropriate framework:

```
putThrowEE :: Members '[AtomicState Bool, Error MyException] r => Sem r Bool
putThrowEE = (atomicPut True *> throw MyException) `P.catch` (\(_ :: MyException) -> atomicGet)
```

We store `True`, throw an error, immediately catch it and return the state. The initial state is always `False` to see whether the state persisted or not. If the result is `True`, we have state-preserving semantics; otherwise, transactional semantics. The full program and output is included in Appendix C.5, here we summarise the results in Figure 7.4.

Figure 7.4: Composing State and Errors: Results and Key

Order / E. System	Raw IO	MTL	EE PS	EE CS
Error inner	True	Right True	Right True	Right True
State inner		Right False	Right False	Right True

EE	Extensible Effects
PS	Pure Semantics
CS	Concurrent Semantics
True	State Preserved
False	State Discarded

There is only one way to write the raw `IO` program—sequentially, with state preserving semantics. Once side effects are emitted, we cannot reverse them. The semantics are unsurprising when errors are the innermost effect since it is equivalent to `(Int, Maybe a)`. When *state* is innermost, MTL and EE PS exhibit transactional semantics. Whether this is an intrinsically desirable behaviour or simply a symptom of state discarding behaviours in traditional effect systems has been disputed [48]. Regardless, the highlighted EE CS cell is undoubtedly inconsistent. It defies our intuition—it is equivalent to `Maybe (Int, a)` and thus we would expect state to be discarded. This behaviour can be viewed as the handler *semantics* winning over the handler *ordering* in the semantic tug of war. That the presence of concurrency reverses the expected compositional semantics is unfortunate, leaking implementation details of `IORef`/`IO` as the source of concurrency. Helpfully, polysemy highlights the quirk in its documentation. However, it is not unique: most extensible effect systems exhibit the same behaviour. King raises similar concerns about other effect compositions translated into extensible effect systems and MTL [48]. Nondeterminism interacts in surprising ways with both `Error` and `Writer`. Therefore, we consider this a criticism of EE systems in their current form.

Higher-order and Control Effects. In §6.4.2 we discussed two orthogonal problems concerning operations that contain monadic programs in negative positions (higher-order effects or HOEs) and those that modify control flow (control effects). Representing these effects is a challenging problem and one still being explored in the effect system design space. Therefore, we will take a pragmatic perspective. The operations in this category employed by a software engineer in typical applications can be counted on one hand. The error operations of **A1** : `throw` and `catch`. Resource management through `bracket` and its derivatives. Less frequently, `local` of `Reader`. How can we express these using extensible effects?

At the centre of the solution provided in the MTL ecosystem is the `MonadBaseControl` type class which enables lifting operations written in terms of the *base* monad (typically `IO`) into an arbitrary stack. Abstractly, the key issue is that we must inevitably go back and forth between the base monad `b` and the target monad `m`. To retain comprehensible program semantics, monadic state must be preserved throughout each of these transitions.

This problem is intrinsic to HOEs, so extensible effect systems need their own solutions. In 2014, Wu et al. proposed one in the context of algebraic effects [117]. This is characterised by a `weave` function used to *interpret* initially embedded HOEs. Several extensible effect frameworks employ this technique; it is best explained by the author of polysemy [123]. Using this construction, the freer monad and further tricks, it is possible to abstract all the complexity behind simplistic APIs. In polysemy, this takes the form of the `Final` GADT for embedding HOEs using a domain-specific language built upon the weave construction [117]. This is a powerful effect and is discouraged in application development.

More importantly for us, ubiquitous HOE operations are encapsulated by `Resource`, `Reader` and `Error` with accompanying interpreters. Thus, the extent to which we are exposed to these complexities is limited to trading, for example, a `Resource` effect for `Final IO`. Therefore, from the perspective of a working functional programmer, the encapsulations of HOEs provided by *monad-control* [101] and *rio* [75] are broadly equal in expressiveness to the polysemy representation. This is an open problem: more recent EE libraries such as *in-other-words* include novel implementations to improve user experience when embedding custom HOEs [124].

***B2**. Modularity* Our analyses of modularity have revolved around the ability to encapsulate parts of our applications using modular abstractions. This is applicable to both effectful programs and interpreters. Using transformer-heavy approaches, like §5.2, this is unfortunately impossible to achieve. We are forced to choose between either modularity in the effectful program *or* in the interpretation. Specifically, concrete transformers demand explicit lifting in our business logic, which is inherently anti-modular. Abstract monads with class constraints trade this problem for the n^2 instances problem: effects can impact other unrelated effects through the type class instances.

Adopting the ReaderT pattern, frameworks such as *rio* encourage the use of a concrete interpreter by providing user-friendly, opinionated operations which are *safe by construction*. However, we are not forced to: the example of §6.3 chooses to continue using an abstract monad. Combined with techniques such as tagless final for custom effects and `Has` for granular environment access, we can achieve modularity in *both* dimensions. This solves the expression problem [60] and mitigates the modularity issues summarised by **C5**, **C6** and **C7**.

The pattern described by the *DTalC* framework constitutes a half-way house: initially embedded DSLs (syntax) with transformer-based interpreters (semantics). Rather than using `Has`, we achieve modularity through an open union `:+:` and the *subtyping* operator `:<:`. This allows us to describe precisely the operations a function may perform, perhaps including user-defined data types to represent domain-specific operations. We *do* make use of `Has` in the *interpretation*, and herein lies the problem. Unfortunately, the evaluation technique described by *DTalC*, using an `Eval` type class and function to fold over the free monadic structure, is not modular nor extensible *in general*. The underlying issue is that `Eval` leaks implementation details of the concrete interpreter. To provide an intuition:

```
class Functor f => Eval f where
  evalAlgebra :: f Int -> Int
```

```
class Functor f => Eval f m where
  eval :: f (m a) -> m a
```

On the left is the original *DTalC* characterisation; on the right our evaluator from §7.1.2. These two evaluators are similar *in structure*. The first captures that every term in the calculator DSL (Hutton's razor [88]) will evaluate to an `Int`. In the case of our more complicated program, the interpreter is a monad `m`—eventually `App`. Both of these make assumptions about the *structure* of the interpreter—an integer and a monad. Kiselyov exposes this problem by attempting to extend the *DTalC* calculator DSL with failure [106].

We could also consider the interpretation of `State` in our free monad implementation. One might reasonably expect to be able to define an `Eval` type class in a manner consistent with the other instructions. Then we would call `eval` with `trackSpeeds` to eliminate state from the coproduct. However, this is not possible. The key problem is that our desired semantics—the *pure* state semantics—requires us to pass the evaluator an initial state. This is inconsistent with its structure. We would have to modify the class itself; leaking implementation details into the abstraction. Instead, we were forced to define a standalone interpretation function: `execState` of Figure 7.1. This suffers from several issues. The order of `State` and the remaining DSL `f` is significant both for the implementation and the call site. `State` must appear on the same side as in the return value of `trackSpeeds`. Moreover, we must manually project by pattern matching `InL` / `InR`; coupling the implementation to the type is undesirable.

Compare this signature with that of `execAtomicStateViaState` in Figure 7.3. Removing the `Functor` constraint and ignoring specific naming, these interpreters of the state effect are *identical*. This overlap is no coincidence. The lack of modularity and extensibility in the *DTalC* interpreter was part of the inspiration of Kiselyov et al. in their original work on extensible effects [22, 23, 106]. The key insight is the use of a type-indexed list for ignoring the order of effects until we pattern match on it. As a result we have an intuitive, modular and extensible pattern for introducing and eliminating effects. This exercise reinforces the argument that extensible effects subsume *DTalC*. Although the two frameworks are built upon the same foundations, extensible effects provide a more native experience and consequently embody the most compelling pattern for embedding initial DSLs.

***B3**. Extensibility* We outline the extension recipes for *DTalC* §7.1.1 and extensible effects §7.3.

The Data Types à la Carte Extension Recipe.

1. Add a (G)ADT with a functor derivation: `FetchLocation (Location -> f)`
2. Add syntax to inject the DSL into a coproduct and `Free`.
3. Provide an instance of `Eval` to interpret the instruction.
4. Include the instruction in the coproduct used as the overall interpreter: the `Effects` alias.

The Extensible Effects Extension Recipe.

1. Add a plain GADT: `FetchLocation m Location`
2. Add syntax to inject the DSL into the extensible effect monad.
3. Write an *effect handler* for interpretation: `runFetchLocation`

We have established that *DTalC* has inherent limitations as a *general* effect system and suggested that the pattern is subsumed by extensible effects. Explicitly enumerating the extension recipes only strengthens this position. The replacement of `Eval` with handler functions is not only an improvement in modularity and extensibility: it also improves type inference and removes boilerplate.

Then, how do extensible effects fare against the tagless final and method dictionary recipes of §6.4.2? On a superficial level, tagless final has the fewest steps, followed by EE and then method dictionaries. Part of this is a result of operations being baked into tagless final type classes, avoiding the need to write (or auto-generate) syntax for operations. More interesting topics of discussion are how we conceptualise the relationships of dependencies for custom DSLs, and the relative merits with respect to the expression problem.

Dependency Graph. In §5.3.2 we introduced the idea of the graph of dependencies when introducing custom capabilities. Central to the expression of these relationships in all three of the previous approaches is the process of *type class instance resolution*. This is most obvious in the Scala example of §6.4.1 since we explicitly declare dependencies for injection with `given` / `using`. Dependencies represented with `Has` projectors and tagless final algebras are parameterised directly by the interpreter. Consequently, the programmer is at the mercy of language mechanics: guiding the compiler to the correct *implicit* resolution. In extensible effect systems, the dependency graph is made *explicit* using functions. This is certainly more direct and arguably more intuitive: introducing and eliminating effects use the same basic language construct. One can look at the running order of handlers to understand the dependencies of different components; though, some might consider it more verbose, enjoying the automation provided by implicit type class resolution.

The Expression Problem in Real World Software. We have a final say on the expression problem, with a particular emphasis on its relevance in software engineering contexts. It is important to highlight that, although the expression problem is a purely academic characterisation, it gives a good indication of the efficacy of a particular language or pattern in real world software development. We can view the issues of modularity and extensibility as *proxies* for practical concerns such as maintainability, readability and testability. If a particular system is feared as a brittle and sensitive to breaking changes, it is more difficult to extend, taking more time and inducing greater costs to the employer.

With this in mind, how might we extrapolate our analyses of pure functional effect systems to software engineering problems? The naive approach of §5.2 is clearly inadequate as a general programming pattern. It is easy to misuse, rife with incidental complexity and cumbersome, with unsatisfactory trade-offs demanded of the programmer at every turn. The techniques described in Chapter 6 emerged directly from conventional wisdom gleaned from real world application of pure functional programming. Therefore, the ReaderT and tagless final patterns satisfy the properties above by their very construction. Using method dictionaries for custom effects and dependency injection is less extensible than both tagless final and extensible effects—this is **D2**. Arguably, however, it has a lower barrier to entry by consisting of only basic language constructs: functions embedded into and projected from data types.

Although it was initially presented using Hutton's razor (a toy example), Swierstra's free monad pattern lead to an increase in adoption of initial embeddings for solving software engineering problems [88, 70]. This came in waves of popularity in different communities: Haskell with its archetypal early adoption, followed by OCaml and later Scala. The different techniques involved are often conflated which can lead to confusion and misuse. Armed with the learnings from our analyses of effect systems, there are a number of recommendations we can make with respect to initial embeddings. We delineate between problems that require extensibility and those that do not.

Inextensible Domains. Certain problem domains have a well established model and are particularly good candidates for DSL-driven development. For example, the *tar* archiving algorithm can be broken apart into a clear instruction set: writing the header, content and footer. This DSL has been implemented in production systems, using streams for interpretation [125]. A better known example is *doobie*, a Scala JDBC wrapper with postgres support [126]. Technologies such as this tend to define explicit protocols, making context free DSLs a particular appealing implementation strategy. For these domains, a vanilla sum type can be perfectly adequate. If the DSL additionally requires type-safety, a GADT might be more appropriate. These constructs are suited to situations in which the instructions are unambiguous and unlikely to change. The role of the free monad is to *sequence* these DSL instructions as programs over the closed union. Moreover, if extension *were* forced upon us, we might consider the compiler guiding us to the relevant interpreters to be a *positive* attribute—rather than indicative of a lack of expressiveness as in Wadler's academic characterisation [60].

Extensible Domains. Given circumstances in which change is more likely, an alternative to the *final* embedding used throughout Chapter 6 is the ad-hoc coproduct approach for composing independent *initial* eDSLs [70, 110]. This allows extensibility in the *syntax* of programs through an *open* union—`trackSpaceStation` of §7.1.2. Again, free monads are simply the sequencing tool. That this extensibility does not extend to the interpreter (established in §7.4.2) may be acceptable if there is only one interpretation; as is often the case in practice. If we would like extensibility baked into a *general purpose effect system*, with native support for standard effects and a higher-level API, then EEs embody a more compelling solution.

***B4**. Ergonomics* Even if all of our other criteria were equal by some objective measurement, ergonomics would inevitably reintroduce subjectivity since it is specifically concerned with the user impact. Therefore, it is important to evaluate the two new effect systems against the MTL-style on the same points of contention. Since several of the compendium entries in §6.5 are related to ergonomics, we interleave them throughout our discussion.

Boilerplate. There is a notable lack of boilerplate code in the EE implementation of §7.3. Given that extensible effects provide a native API, abstracting the functor and coproduct elements of *DTalC*, there is a natural reduction in boilerplate. The three custom effects require only syntax for injection into the effect system. These can be automated through meta-programming. Although the same can be done with the `Has` instances associated with the ReaderT pattern, there are two stages to automation: generating instances for the concrete environment, and creating syntax for projection. One distinctive advantage is that we can forgo the `MonadReader` constraint for accessing dependencies embedded as method dictionaries. The `Reader` effect is used for its intended purpose: dynamic binding. Therefore, just as §6.3 reduced the amount of boilerplate necessary in §5.2, we have again made an iterative improvement towards a solution with minimal incidental complexity. One might consider *explicit* handlers more verbose than tagless final interpreters; enjoying the automation provided by *implicit* type class resolution. Nevertheless, we have reduced the boilerplate caused by the n^2 instances problem (**C6**), explicit lifting (**C7**), and the `Has` pattern (**D1**).

Type Inference. **D7** summarised two sources of type inference problems: type class instance resolution of tagless final eDSLs, and duplicate `Has` constraints in effectful programs. The former is solved by the elimination of type classes as an integral part of the effect system's mechanics. Instead, we use only functions and data. For the latter, the ability to duplicate effects in the open union mitigates this issue. If type inference fails we can provide an annotation such as `runError @ISSError`. More generally, the use of a type-indexed coproduct as the representation of an open union is inherently more precisely typed than class constraints / instances. Therefore, it is less prone to type inference issues. We have also resolved the type inference issues encountered during the free monad implementation §7.1.2; caused by the custom-made coproduct and overlapping instances for the subtyping operator. Polysemy encapsulates these details in the `Sem` monad.

Cognitive Overhead. It is uncontroversial to state that both approaches deploying initial DSL embeddings are less cognitively challenging than modification and extension of a transformer stack. We have another viable solution which overcomes the usability issues caused by the n^2 instance problem: captured in **C6**, **C8** and **C10**. A more contentious quandary is whether extensible effects are more comprehensible than the tagless final pattern: the premise of **D6**. Just as tagless final can be used in practice without having a deep understanding of its academic origins or the relevant language constructs, EEs may also be applied without extensive background knowledge. For instance, it is not necessary to understand the type-indexed coproduct or freer monad constructions to employ an extensible effect framework. DSL descriptions are data types. Interpreting instructions is a regular function, and is consistent with how we *introduce* effects. For the complex elements, drawing comparisons with more fundamental functional concepts can ease this burden: instead of pattern matching on lists of data, we are pattern matching on a *type-level list* of effects. Type classes and type class instances are also common idioms applied to a particular use case—propagating effect operations. Hence, it is difficult to claim that one is more conceptually challenging than the other. Regardless, it will be important to continue to improve the developer experience off EE systems through library APIs, documentation, guides and blog posts.

Method dictionaries arguably have a lower entry barrier than both these approaches. They consist of more basic language constructs: functions embedded into and projected from data types. The use of concrete types, accepting the

modularity trade-offs, is an additional simplification. In this regard, the ReaderT pattern and frameworks like rio are appealing [76, 93].

Part of the cognitive cost of a programming pattern is its *consistency*. **D5** argues that the design choices made throughout Sections 6.3 and 6.4 are ad-hoc and inconsistent: working around fundamental limitations of transformers. If this were the sole criterion then even the naive MTL example would have some appeal. The use of transformers for *every* effect constitutes a well-defined pattern which, once grasped conceptually, can be applied with knowledge of the numerous pitfalls. Conversely, tagless final, the ReaderT pattern, method dictionaries, pure mutable references and `IO` exceptions are orthogonal ideas. Our simple ISS example illustrated how to combine these and induce a compelling programming pattern. However, when scaled to larger applications, this lack of consistency can lead to a greater maintenance burden. Consider viewing for the first time a codebase that deploys some subset of the techniques described in Chapter 6. Since there are many variants, understanding program behaviour and the expression of particular effects may require a deeper investigation. Moreover, teaching someone less experienced carries baggage. Admittedly, opinionated frameworks lower this barrier to entry by establishing consistent patterns for common use cases.

Extensible effects are closer to the example of §5.2 than that of §6.3. There is a consistent mechanism for expressing both our four key effects **A1**–**A4** and more advanced operations. More importantly, extending a program with *custom effects* is also consistent: write a GADT and syntax for effect construction, then a function interpreter for effect deconstruction. This simplicity is appealing. Of course, the nuances of higher-order and control effects remain; these are inherently subtle. Hence, a library ecosystem using similar patterns is important for EE systems to thrive.

Susceptibility to Errors. It is clear that extensible effect systems reduce a programmer's propensity to make errors when compared with the transformer-heavy approach, addressing at least **C6** and **C9** of those negatives. There is an argument that the higher level of abstraction and consistency of frameworks such as polysemy would induce fewer mistakes than an ad-hoc ReaderT based solution. Different codebases may employ different patterns and perhaps different libraries. Under harsher scrutiny, we may question whether such a programming style can be considered an effect system at all; it is more similar to the targetted language level constructs of imperative languages. A more fully-fledged, opinionated, and safe solution like rio is likely to be more robust [93, 75]. This a valid alternative to EE frameworks.

7.4.3 Performance To begin, we reiterate our reasons for relegating performance as a primary criterion, having covered the necessary prerequisites:

1. The costs of reduced efficiency are dwarfed by those associated with unmaintainable and brittle systems. Maximising the composition criteria **B1**–**B4** contributes significantly towards minimising such costs.

2. Benchmarks are challenging to get right and can be misleading, as illuminated by the authors of eff, polysemy and Cats Effect in Scala [85, 127, 128]. We should of course still measure performance characteristics, but they should not typically be the deciding factor in choosing a particular effect system.

3. The majority of industry software applications are I/O-bound rather than CPU-bound. Consequently, higher-level tests are more practically significant than framework level benchmarks *regardless* of their efficacy.

4. Give that we *do* have a compelling solution with respect to our criteria, then we can push the boundaries of efficiency—hacking optimisations on top if necessary.

With this in mind, we address the issue of performance across the full spectrum of the effect systems we have covered. In general, it is difficult to measure the performance characteristics of programs in a meaningful way. Black box system-level load tests can give us a high-level understanding of how our systems may perform with respect to throughput, responsiveness and resource usage. At the other end of the spectrum, micro-benchmarks can give us specific metrics about compiler optimisations and the efficiency of different language features.

Effect System Performance. When evaluating the efficiency of an *effect system*, neither of these are particularly insightful. The effect systems we have covered consist of different blends of language constructs, data structures and chained operations. Therefore, high-level tests are broad, but too far away from the metal to evaluate subtle performance metrics and remedy bottlenecks. Micro-benchmarks are too granular and not representative of the types of applications ubiquitous in industrial development. Moreover, it is difficult to isolate the effect system performance among the noise of domain-specific computations. King performed a comprehensive assessment of Haskell performance, sharing the results in a conference talk [85]. Given the subtleties involved, we refer to this useful resource throughout the following evaluation.

Since the introduction of transformers by Liang et al. [2], Haskell has steadily gained popularity in both academic and industrial settings. Consequently, the efficiency of its characteristic features have been scrutinised. This includes but is not limited to: the `IO` monad, monadic sequencing (`>>=`), data types, type class mechanics, pattern matching and evaluation semantics. By virtue of referential transparency, there are a number of optimisations a compiler can

make in a pure functional language. These include the in-lining of function calls, application of reduction rules, fusing of data structures, and specialisation of abstractions. All of these constructions and optimisations are relevant to the performance of effect systems.

Recently, benchmark results given by Haskell effect frameworks were exposed as highly misleading [85]. They overfit to the benchmarking situations and are therefore too reliant on these optimisations being triggered. This is less likely in larger, multimodule applications which make use of abstractions and combine several language features. Therefore, running our own benchmarks against the ISS examples would not be particularly illuminating. We can, however, discuss in *general* terms the performance characteristics of the frameworks we have covered.

Transformers. With these caveats we assess the landscape of effect systems with respect to the topic of runtime performance[1], starting with MTL. Throughout the analysis of transformers we have discussed the trade-offs involved when choosing between a concrete transformer or an abstract monad with type classes. In terms of performance, the former is the undisputed winner. Monad transformers are generally `newtype` wrappers around functions and are therefore highly amenable to in-lining of operations such as `>>=` ; they are almost a "zero-cost abstraction" [85].

Unfortunately, the use of MTL-style type classes for effects triggers a different compiler path, violating the conditions under which these optimisations can fire. We can view type classes as wrappers around one or more named functions, like the environment-embedded capabilities of the ReaderT pattern. Indeed, the Haskell compiler takes this view: type classes are passed using *records of functions*, often called *dictionary passing*. This precludes many of the optimisations we listed: higher-order functions cannot be in-lined since they constitute *unknown calls* [85]. As King illuminates, the reason that MTL typically outperforms EE systems in benchmarks is due to the *specialisation* process. For single-module programs, the Haskell compiler can *monomorphise* abstract type parameters into the concrete types used at call-sites. This can trigger aggressive in-lining, leading to fast benchmark results. Real world programs are rarely ever single-module. Thus, programs making heavy use of abstractions, such as the abstracted naive MTL example (§5.3.2) and the tagless final implementation (§6.3), are inevitably slower than concrete transformers.

However, this is not *always* the case. There is a performance penalty for each layer in a transformer stack. This cost is *pay-per-effect* [22]: using a specific transformer operation once in a program forces *every* operation through the extra layer. This renders our transformer-heavy naive implementation of §5.2 even less appealing. Hence, this is another point in favour of the ReaderT pattern and frameworks such as rio; they contain only a single transformer which can be used concretely, maximising theoretical performance.

Initial Embeddings. Free monads are historically known to be slow. Intuitively, we are building tree-like data structures and peeling off instructions during interpretation. Using a naive encoding such as *DTalC* [70] has a significant memory footprint and similarly high garbage collection cost—wasting precious CPU cycles. Furthermore, that these are *embedded* DSLs precludes several compiler optimisations since our programs are written using domain-specific data constructors rather than native, host language types.

Thankfully, several optimisations exist based on academic innovations. Shortly after Swiestra's *DTalC*, Voigtländer made a stark improvement to the performance of free monads [70, 131]. This led to more efficient encodings such as Kmett's *Church encoding*, provided in the popular *free* library [132, 133]. These are achieved by applying equational laws to collapse calls to `fmap` and `>>=`.

Thanks to the freer monad construction, extensible effect systems also have methods for improving performance [23]. Making the monadic continuation an *initial* embedding in `FEFree`, rather than a pre-composed sequence of functions, allows the removal of parts of the sequenced program. Moreover, the work of Wu et al. yielded efficiency gains using *fusion laws* to avoid constructing intermediate representations of effects [61]. These techniques are employed by the fused-effects and polysemy libraries to improve their performance characteristics [108, 109].

Regardless of these enhancements, the current best effort EE system will not compete with MTL programs on micro-benchmarks due to the lack of specialisation and subsequent in-lining opportunities. As the domain currently stands, the best option for maximal performance would be a concrete ReaderT-based representation of effects, such as the rio framework [93, 75]. We are not afforded the option of concrete representations in extensible effect systems as a result of the *necessarily abstract* type-indexed list.

We should also consider the potential for *future* performance gains. Effect handlers are layered in a different way to transformers. Effects propagate up the layers until they find a handler. Once eliminated, the overhead of that effect is removed. In other words, each effect is *pay-per-use* rather than MTL's *pay-per-effect* [22]. Therefore, the theoretical performance ceiling of extensible effects may be higher than that of transformers. Of course, the *most* efficient representation of algebraic or extensible effects would be one residing in a language with *native* support.

Hybrid Languages. What is the situation in hybrid languages such as Scala? Compilers of impure languages are fundamentally different. Due to the lack of referential transparency, the kinds of optimisations available to Haskell are off the table. Since the JVM supports just-in-time compilation, as opposed to Haskell's ahead-of-time compilation, the

[1] Considering modern techniques for incremental compilation, languages servers such as HLS, metals, and fully-fledged IDEs, *compiler* performance is considered inconsequential [129, 130].

runtime can react to the *hot paths* taken by programs, adapting accordingly to maximise efficiency. However, these languages are inherently less specialised and not designed with pure functional programming constructs in mind. It is the responsibility of library authors to exploit particular language features and squeeze the best performance out of the runtime.

The performance problems of transformers in Scala have arguably been overstated—though, for reasons we have covered, they are rendered broadly unnecessary for production systems and subsumed by other techniques. Applying our recommendations of §6.1 to §6.4.1 in Scala, there is not the same stark performance difference between an abstract and concrete effect monad; due to type classes instances being passed as *implicit parameters* rather than method dictionaries. The performance of different effect monads is contested, with different frameworks claiming to be the fastest. Spiewak gives a comprehensive and balanced exposition, similar to King [85], on the subtleties involved in both pure functional benchmarking and performance on the JVM more generally [128, 134]. Suffice to say, the differences are rarely consequential—particularly for I/O bound applications. If a particular use case demands high performance, one can make domain-specific optimisations or utilise the characteristic flexibilty of hybrid languages to speed up the most important code paths.

7.5 Compendium

We perform a final summary of the analysis, including references to the previous compendium.

Positives

1. *Environment & Ergonomics*. Compared with the best we have of MTL-style programming, boilerplate is minimised as much as possible without having first-class language support for extensible effects. We can consider **D1** solved.

2. *Expressiveness*. **D3**, the most important remaining issue of expressiveness is resolved: semantics are no longer fixed. We can now repeat effects and express different semantics within a single program or *function*.

3. *Expressiveness*. We are less exposed to the complexities of higher-order and control effects captured by **D4**. Extensible effect frameworks provide native encapsulations which are more intuitive than the corresponding MTL type classes.

4. *Expressiveness & Ergonomics*. Holistically, extensible effect systems have more consistent idioms for representing the syntax and semantics of effects than the ad-hoc solutions epitomised by the recommendations of the ReaderT pattern in §6.2. **D5** is solved.

5. *Modularity & Extensibility*. Extending a system with a new effect is at least as seamless as the tagless final approach of §6.1. We have another solution to the expression problem.

6. *Ergonomics*. Using a type-indexed coproduct as the open union instead of type class constraints results in better type inference; specifically when compared with extensive use of tagless final and `Has`.

Negatives

1. *Expressiveness*. Certain effect operations have unexpected semantics; a product of leaking lower-level implementation details.

2. *Expressiveness & Ergonomics*. Expressing custom higher-order effects in a user-friendly manner is challenging and remains an open research area.

3. *Ergonomics*. Manually defining operation syntax still constitutes boilerplate. Macros are as good as we can get without wholesale language changes to provide native extensible effects.

4. *Ergonomics*. As with the tagless final barrier to entry, summarised by **D6**, extensible effect systems have complex elements. Languages with first-class support for extensible/algebraic effects will inevitably have better usability.

5. *Performance*. Although performant enough for most practical situations, the fastest extensible effect systems available today are slower than the equivalent programs expressed using concrete transformers.

Many of these negatives relate to the usability of extensible effect systems. This is likely to improve since, relative to MTL, the domain its still in its infancy. The resolution of the expressiveness problems inherent to transformers is a powerful advantage. Even more *practically* important are the issues of modularity, extensibility and the consistency of the framework as a whole. Judging by our analysis, EE systems are unrivalled in these measures. There are several promising avenues for future work, which we enumerate in the conclusion.

8 Conclusion

We have traversed the domain of pure functional effect systems with the objective of understanding the relative merits of various tools and design patterns. This has given us a solid grounding in both the general concepts and specific techniques. It will be helpful to aggregate our findings—a compendium of compendiums. We depart from the more abstract criteria sets of **A1**–**A4** and **B1**–**B4**, instead presenting these tools as solutions for specific classes of software engineering problems. We use our contributions as a guide, before identifying avenues for future work.

8.1 Contributions

Contribution 1: Recommended Application of Pure Functional Effect Systems The primary goal of this project was to evaluate pure functional effect systems from first principles through a practical lense. Having analysed the relevant trade-offs, we now present our findings as a set of recommendations regarding when to apply the tools in the toolkit of a working functional programmer. We can view our results in two dimensions: a problem-first approach for practical application, and a tool-first approach to develop understanding.

Software Engineering Problems. Given a problem, which tool or technique is most appropriate?

- *General Application Development*. Extensible effects embody the most compelling effect system for typical, I/O bound application development. ReaderT-based solutions are less consistent but also viable. Safe, opinionated frameworks like rio are an appealing alternative [75]. However, they do not have the same native support for DSL-driven development. Free monads can be a useful implementation strategy for particular components with well established models. For such applications, the performance characteristics at the level of the *effect system* are likely to be inconsequential for *overall* performance.

  In Scala, extensible effects have not had the same attention. Thus, the current best approaches are either tagless final with a classic effect monad such as Cats Effect `IO`, or a more opinionated effect system like `ZIO` [51, 52]. Evaluating these two leading approaches in pure functional Scala requires its own detailed analysis; we leave this for future work.

- *Smaller Programs*. There is a class of programs with a smaller scope, for instance: internal tools, short-lived applications written for a particular purpose, and small serverless programs. Extensibility is not a primary concern in these situations. Therefore, it may not be necessary to commit to a fully-fledged effect system: a more ad-hoc combination of our tools can be satisfactory. For instance, a custom interpreter providing operations through MTL classes, tagless final and method dictionaries, or a solution based on vanilla free monads. Extensible effects are still applicable, but require knowledge of a particular framework.

- *High Performance Applications*. If using pure functional programming for high performance applications—constrained by CPU and memory—the fastest option is inevitably a raw effect monad such as `IO`. For better modularity and maintainability, a concrete ReaderT-based solution is a strong alternative. In Scala, avoid transformers if performance is a priority.

- *Software Libraries*. Libraries often attempt to abstract particular technologies or usage patterns. Thus, DSL-driven development is a useful implementation strategy, so initial DSL embeddings constitute an appealing tool. Moreover, extensible effects provide many opportunities for user-friendly, abstracted instruction sets. Therefore, developing an ecosystem of libraries built around a particular extensible effect framework is important for improving user experience and would lead to increased adoption.

Toolkit. Given a particular tool or technique, which problems is it suited to?

- *Monads*. The programmable semicolon, used for sequencing computations.

- *Monad Transformers*. Data structures for vertically composing more than one monadic effect. Useful to gain access to effect operations in smaller software components. Composition of several transformers is discouraged due to the lack of extensibility, performance cost, susceptibility to errors and broader usability issues. Hence, avoid as a general effect system.

- *MTL-style Programming*. a set of tagless final eDSLs for standard effects. Can be used as abstractions independently of transformers. Custom capabilities can be written using the same technique.

- *The ReaderT Pattern*. A set of recommendations centered around the ReaderT monad transformer with its safe compositional semantics. Solves many of the usability concerns afflicting transformers, but inherent limitations regarding expressiveness persist. Perhaps too ad-hoc to be called an effect system. Nevertheless, a compelling programming pattern for general application development; aided further by opinionated frameworks. Has strong performance characteristics when used concretely, at the cost of modularity.

- *Method Dictionaries*. Embedding dependencies into the environment as a record of functions. Typically used in conjunction with the ReaderT pattern.
- *Tagless Final*. A technique for creating embedded domain-specific languages with strong extensibility. Induces an appealing programming pattern when combined with ReaderT.
- *Initial Embedding*. A technique for writing context-free DSLs as (G)ADTs. Suited to domains with well-defined models that are unlikely to change.
- *Coproducts*. A tool for *composing* independent initial DSL embeddings represented as functors using functor composition. Useful for achieving extensibility when employing initial eDSLs.
- *Free monads*. A tool for *sequencing* initial DSL embeddings represented as functors by constructing a tree of instructions. Combined with coproducts, constitutes an extensible pattern for DSL-driven development. Avoid as a general effect system due to lack of extensibility in the interpreter: prefer extensible effects.
- *Freer monads*. A tool for sequencing *GADTs* without the functor constraint. An implementation detail of extensible effect systems. Gives rise to performance optimisations not available to traditional free monads.
- *Extensible Effects*. A native effect system based upon initial eDSLs with built-in composition and standard effects. A compelling tool for general DSL-driven application development. Frameworks require buy-in; thus, free monads may be preferable in smaller components. Cannot yet compete with the performance of concrete transformers.

Contribution 2: Applying Extensible Effects The application and analysis of extensible effects in Chapter 7 aims to bridge the gap between academia and the frameworks used by software engineers: our second contribution. Though it is not necessary to fully understand the inner workings of a framework to make use of it, many developers prefer a more holistic view of design patterns and frameworks. This can be important for evaluating trade-offs when building real world applications. To my knowledge there has not yet been such an exposition. It is hoped that we clarify the overlap between the different concepts and techniques; connecting the programming patterns to the ideas in the literature.

Contribution 3: Scala 3 Dependency Injection A final, smaller contribution is a specific technique for dependency injection when writing pure functional Scala. In §6.4.1, we presented a novel pattern for grouping tagless final style type class constraints in Scala 3; expanded on in Appendix B.3.1. It is based upon *context functions*, a feature not available in previous versions of Scala [97]. This constitutes a useful tool to reduce verbosity and encourage modularity. Usability has room for improvement; meta-programming could provide the answer, at the cost of transparency. Moreover, it can be adapted for contexts broader than pure functional Scala—environment passing is an important tool in all Scala paradigms.

8.2 Future Work

We have mentioned in passing potential avenues for future work. These are now aggregated in one place.

Extensible Effects. As a programming framework, EEs are still in their infancy. Several new libraries are already being developed in Haskell, pushing the envelope on particular areas of the design space. The outcomes of our analyses reinforce that these effect systems are certainly *worth* optimising. Therefore, I feel the following areas of focus are particularly important and have high growth potential: performance, the representation of higher-order effects, and building a library ecosystem. This aligns closely with the criticisms in our final compendium §7.5. I would also echo Diehl's sentiment that language-level support for extensible or algebraic effects in *production-ready* languages would lead to improvements in both user experience and efficiency [135].

Effect Systems in Pure Functional Scala. Though many of the principles are the same, we have only scratched the surface of expressing effects in hybrid languages such as Scala. It would be interesting to perform a similar analysis, perhaps using the same criteria sets, in the context of pure functional Scala. Given the flexibility of such languages, and the differences in runtime characteristics, there is a gap in the design space for extensible and algebraic effect systems—novel approaches are likely to appear as libraries. Moreover, there is a research project currently investigating how to integrate a notion of algebraic effects into the Scala language itself [136]. Thus, recognition of the merits of these effect systems is reaching the design space of more mainstream languages.

Algebraic Effects. In the context of impure functional and imperative languages, algebraic effect systems have a similar appeal to extensible effects. There is also the same gap in the literature regarding the *application* of languages with native algebraic effects to larger problems. The criteria used for our evaluation could be applied to a comparison of algebraic effects with traditional imperative effect representations, in the context of a real world application. Languages with an industry focus such as Unison are promising vehicles for such an exposition [18]. In particular, can a sufficiently powerful algebraic effect system generate many of the same benefits enjoyed by pure functional programmers, while forgoing referential transparency? I plan to explore this in future work.

Testing. We only briefly covered the topic of unit testing in §6.4.2. In industry software, unit and integration tests of software components are an important part of a developer's role. This topic receives less attention in pure functional programming communities than those of imperative languages. This is likely due to extracting as much expressiveness as possible from static typing—the preference for "types over tests". There are also different modes of thinking regarding the most effective testing strategies and corresponding patterns. Being a real world software concern, the topic of testing is unsurprisingly absent from the literature. Therefore, an investigation into the relative merits of these effect systems with respect to testability is an interesting unexplored domain.

A Haskell Code

Full implementations of the International Space Station example, including different variants based on changes suggested in the prose.

A.1 Main

The main program running each implementation sequentially. Mostly just pretty printing: each is marked with a label and number. Full application logs can be found in Appendix E.1.

```
module Main (main) where

import           Control.Monad       (zipWithM_)
import qualified ExtensibleEffects
import qualified ExtensibleEffectsAlternate
import qualified FreeMonads
import qualified FreeMonadsAlternate
import qualified MonolithicMonad
import qualified TransformersImproved
import qualified Transformers

main :: IO ()
main = let examples = fmap (uncurry runExample) labelledExamples
       in zipWithM_ ($) examples [1..(length examples)]

type LabelledExample = (String, IO ())

labelledExamples :: [LabelledExample]
labelledExamples =
  [ ("Monolithic monad", MonolithicMonad.main)
  , ("Transformers, both concrete and abstract", Transformers.main)
  , ("Transformers improved (ReaderT, Has, tagless final)", TransformersImproved.main)
  , ("Free monads", FreeMonads.main)
  , ("Free monads alternate (granular environment + StateT)", FreeMonadsAlternate.main)
  , ("Extensible effects, polysemy", ExtensibleEffects.main)
  , ("Extensible effects, polysemy alternate", ExtensibleEffectsAlternate.main)
  ]

runExample :: String -> IO () -> Int -> IO ()
runExample name example i =
  let dashes = concat $ replicate 15 "-"
      padLength = 68 - length name
      endDashes = concat $ replicate padLength "-"
      space  = " "
  in print (dashes ++ space ++ show i ++ ": " ++ name ++ space ++ endDashes) >> example
```

A.2 Common Code: Model, Logic and Utility Functions

Code shared by all implementations. We omit most of this from the prose since it is not of great significance for the effect system analyses. Broadly in order we have:

1. `newtype` wrappers mostly for readability and avoiding mistakes.
2. Product types declared with `data`. `Config` and `AppData` have named record accessors used in the effectful programs for the environment and state effects.
3. Type class instances, such as `FromJSON` of the *aeson* library, used for parsing.
4. The application configuration value is defined once. `parseRequest_` is unsafe (can throw an exception); we use it only for brevity since it is injected into several programs.
5. Arithmetic for computing the speed: `Data.LatLong.geoDistance` of the *persistent-spatial* library does most of the heavy lifting here.
6. Useful combinators for composing effectful values.

```
module Common where

import Data.Aeson           (FromJSON (parseJSON), withObject)
import Data.Aeson.Types     (parseField)
import Data.Functor         (($>))
import Data.LatLong         (LatLong (LatLong), geoDistance)
import Data.Text            as T (pack)
```

```haskell
import Network.HTTP.Simple (Request, parseRequest_)
import Text.Read           (readMaybe)

newtype Lat          = Lat Double deriving Show
newtype Lon          = Lon Double deriving Show
newtype Count        = Count Int deriving (Eq, Ord, Num, Show)
newtype Timestamp    = Timestamp Integer deriving Show
newtype Seconds      = Seconds Int
newtype KmPerHour    = KmPerHour Double deriving Num
newtype MetersPerSec = MetersPerSec Double deriving Num
newtype Microseconds = Microseconds { getMicroseconds :: Int } deriving Num
type Speeds          = [KmPerHour]
data Config          =
  Config { httpRequest :: Request, totalRequests :: Count, requestDelay :: Seconds }
data Location        = Location Lat Lon Timestamp deriving Show
data AppData         =
  AppData { location :: Location, remainingRequests :: Count, speeds :: Speeds }

instance Show KmPerHour where show (KmPerHour kph) = show (round' kph 2) ++ "km/h"

instance FromJSON Location where
  parseJSON = withObject "Location" $ \topObject -> do
      coordsObj  <- parseField topObject (T.pack "iss_position")
      (lat, lon) <- parseCoords coordsObj
      ts         <- parseField topObject (T.pack "timestamp")
      return $ Location lat lon (Timestamp ts)
    where
      parseCoords = withObject "iss_position" $ \obj -> do
        lat <- parseField obj (T.pack "latitude") >>= parseDouble "latitude"
        lon <- parseField obj (T.pack "longitude") >>= parseDouble "longitude"
        return (Lat lat, Lon lon)
      parseDouble name raw =
        foldl (\_ a -> pure a) (fail $ name ++ " must be a Double") (readMaybe raw)

issRequest :: Request
issRequest = parseRequest_ "http://api.open-notify.org/iss-now.json"

config :: Config
config = Config issRequest (Count 3) (Seconds 5)

round' :: Double -> Integer -> Double
round' d n = let x = 10^n in (fromIntegral . round $ d * x) / x

updateState :: Location -> AppData -> (KmPerHour, AppData)
updateState newLoc (AppData oldLoc remainingReq oldSpeeds) =
  let newSpeed = computeSpeedKmPerHour oldLoc newLoc
  in (newSpeed, AppData newLoc (remainingReq - 1) (newSpeed : oldSpeeds))

secsToMicrosecs :: Seconds -> Microseconds
secsToMicrosecs (Seconds s) = Microseconds $ s * 1000000

metersPerSecToKmPerHour :: MetersPerSec -> KmPerHour
metersPerSecToKmPerHour (MetersPerSec mps) = KmPerHour $ mps * 3.6

timestamp :: Location -> Integer
timestamp (Location _ _ (Timestamp ts)) = ts

computeSpeedKmPerHour :: Location -> Location -> KmPerHour
computeSpeedKmPerHour a b =
  let deltaMeters  = geoDistance (mkLatLon a) (mkLatLon b)
      deltaSeconds = fromIntegral $ timestamp a - timestamp b
  in abs $ metersPerSecToKmPerHour $ MetersPerSec (deltaMeters / deltaSeconds)

mkLatLon :: Location -> LatLong
mkLatLon (Location (Lat la) (Lon lo) _) = LatLong la lo

flatTap :: Monad m => m a -> (a -> m b) -> m a
flatTap fa g = fa >>= (\a -> g a $> a)

tapM :: Monad m => a -> (a -> m b) -> m a
tapM = flatTap . return
```

A.3 Example Implementation: Monolithic Effect Monad

```haskell
module MonolithicMonad (main) where
```

```haskell
import Common                 (AppData (..), Config (..), Location,
                               Microseconds (..), Seconds, Speeds,
                               config, secsToMicrosecs, tapM,
                               updateState)
import Control.Concurrent     (threadDelay)
import Control.Monad          (join)
import Data.Functor           (($>))
import Network.HTTP.Simple    (JSONException, Request,
                               getResponseBody, httpJSONEither)
import Prelude                hiding (log)

type Logger = forall a. Show a => a -> IO ()

main :: IO ()
main = trackSpaceStation print config

-- Effect syntax and semantics are indistinguishable, modular abstraction is minimal
trackSpaceStation :: Logger -> Config -> IO ()
trackSpaceStation log (Config request initialCount delay) = do
  maybeInitialLoc <- fetchLocation log request
  maybeSpeeds     <- fmap join (mapM loop maybeInitialLoc)
  either logFailure logSuccess maybeSpeeds
  where
    loop initialLoc = trackSpeed log request delay (AppData initialLoc (initialCount - 1) [])
    logSuccess finalSpeeds =
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure e = log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: Logger -> Request -> Seconds -> AppData -> IO (Either JSONException Speeds)
trackSpeed log request delay state =
  if remainingRequests state <= 0
  then log "No more requests to send to ISS" $> Right (speeds state)
  else do
    threadDelay $ getMicroseconds $ secsToMicrosecs delay
    maybeLoc <- fetchLocation log request
    either onErrorReturn onSuccessLoop maybeLoc
  where
    onErrorReturn = return . Left
    onSuccessLoop currentLoc = do
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      trackSpeed log request delay newState

fetchLocation :: Logger -> Request -> IO (Either JSONException Location)
fetchLocation log request = do
  maybeBody <- httpJSONEither request
  mapM (`tapM` log) (getResponseBody maybeBody)
```

A.4 Example Implementation: Monad Transformers, Naively

```haskell
module Transformers (main) where

import Common                           (AppData (..), Config (..),
                                         Location, Microseconds (..),
                                         config, flatTap,
                                         secsToMicrosecs,
                                         updateState)
import Control.Monad.Except             (ExceptT, MonadError,
                                         catchError, liftEither,
                                         runExceptT, void)
import Control.Monad.IO.Class           (MonadIO)
import Control.Monad.Identity           (IdentityT (runIdentityT))
import Control.Monad.Logger             (LoggingT, MonadLogger,
                                         logInfoN, runStdoutLoggingT)
import Control.Monad.Reader             (MonadReader (reader),
                                         ReaderT (..))
import Control.Monad.State              (MonadState (get, put),
                                         StateT, execStateT)
import Control.Monad.Time               (MonadTime (threadDelay))
import Control.Monad.Trans.Class        (MonadTrans (lift))
import Control.Monad.Trans.SimulatedTime (RealTimeT (..))
import Data.Function                    ((&))
import Data.Functor                     ((<&>))
import Data.Text                        as T (pack)
import Network.HTTP.Simple              (JSONException,
                                         getResponseBody,
```

```
                                        httpJSONEither)
import Prelude                         hiding (log)

main :: IO ()
main = do
  interpret trackSpaceStation
  interpret trackSpaceStation'

-- Effect operations: syntax / construction / introduction
-- All other effect operations are used directly from external libraries.
class Monad m => MonadSpaceStation m where
  fetchLocation :: m Location
  default fetchLocation :: (MonadSpaceStation m', MonadTrans t, m ~ t m') => m Location
  fetchLocation = lift fetchLocation

log :: (Show a, MonadLogger m) => a -> m ()
log a = (logInfoN . T.pack) $ show a

-- Concrete transformers: effectful program composing multiple effects / capabilities
trackSpaceStation :: App ()
trackSpaceStation = catchError program logFailure
  where
    program = do
      initialCount <- reader totalRequests
      initialLoc   <- fetchLocation
      finalSpeeds  <- execStateT trackSpeed (AppData initialLoc (initialCount - 1) []) <&> speeds
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure e = log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: StateT AppData App ()
trackSpeed = do
  state <- get
  if remainingRequests state <= 0
    then log "No more requests to send to ISS"
    else do
      reader requestDelay >>= threadDelay . getMicroseconds . secsToMicrosecs
      currentLoc <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      put newState
      trackSpeed

-- Abstract monad: effectful program composing multiple effects / capabilities
type MonadAppCommon m =
  (MonadSpaceStation m, MonadReader Config m, MonadLogger m, MonadTime m)

trackSpaceStation' :: (MonadAppCommon m, MonadError JSONException m) => m ()
trackSpaceStation' = catchError program logFailure
  where
    program = do
      initialCount <- reader totalRequests
      initialLoc   <- fetchLocation
      finalSpeeds  <- execStateT trackSpeed' (AppData initialLoc (initialCount - 1) []) <&> speeds
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure e = log $ "Failed to call ISS with error: " ++ show e

trackSpeed' :: (MonadAppCommon m, MonadState AppData m) => m ()
trackSpeed' = do
  state <- get
  if remainingRequests state <= 0
    then log "No more requests to send to ISS"
    else do
      reader requestDelay >>= threadDelay . getMicroseconds . secsToMicrosecs
      currentLoc <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      put newState
      trackSpeed'

-- Interpreters for effects and application stack: semantics / deconstruction / elimination
newtype SpaceStationT m a = SpaceStationT { runSpaceStationT :: IdentityT m a }
  deriving (Functor, Applicative, Monad, MonadReader r, MonadError e, MonadLogger,
            MonadTime, MonadIO, MonadTrans)

instance (MonadReader Config m, MonadError JSONException m, MonadLogger m,
```

```
          MonadIO m) => MonadSpaceStation (SpaceStationT m) where
  fetchLocation = SpaceStationT $ do
    maybeResponse <- reader httpRequest >>= httpJSONEither
    (liftEither . getResponseBody) maybeResponse `flatTap` log

newtype App a = App {
  runApp :: RealTimeT (SpaceStationT (ReaderT Config (LoggingT (ExceptT JSONException IO)))) a
} deriving (Functor, Applicative, Monad, MonadIO, MonadLogger, MonadReader Config,
            MonadError JSONException, MonadSpaceStation, MonadTime)

-- Tear down the stack of transformers and discard the result
interpret :: App a -> IO ()
interpret program =
  runApp program
    & runRealTimeT
    & runSpaceStationT
    & runIdentityT
    & (`runReaderT` config)
    & runStdoutLoggingT
    & runExceptT
    & void

-- Transformer boilerplate to propagate effect interfaces to the top of the stack
instance MonadTrans RealTimeT where lift = RealTimeT
instance MonadLogger m => MonadLogger (RealTimeT m)
instance (MonadSpaceStation m) => MonadSpaceStation (RealTimeT m) where
instance (MonadSpaceStation m) => MonadSpaceStation (StateT s m) where
```

A.5 Example Implementation: Monad Transformers, Pragmatically

The full example discussed in §6.3.

```
{-# LANGUAGE DerivingVia #-}

module TransformersImproved where

import Common                 (AppData (..), Config (..), Count,
                               Location, Microseconds (..), Seconds,
                               config, secsToMicrosecs, tapM,
                               updateState)
import Control.Concurrent     (threadDelay)
import Control.Monad.Catch    (MonadCatch (..), MonadThrow (throwM),
                               SomeException, Exception)
import Control.Monad.IO.Class (MonadIO, liftIO)
import Control.Monad.Reader   (MonadReader (reader), ReaderT (..))
import Control.Monad.Ref      (MonadRef, newRef, readRef, Ref, writeRef)
import Data.Functor           (($>), (<&>))
import Data.Has               (Has (getter))
import Network.HTTP.Simple    (Request, getResponseBody, httpJSON)
import Prelude                hiding (log)


main :: IO ()
main = let logger  = Logger (liftIO . print)
           sleeper = Sleeper (liftIO . threadDelay . getMicroseconds)
       in runApp trackSpaceStation $ Env config logger sleeper

-- Effect syntax (construction / introduction)
class Monad m => MonadSpaceStation m where
  fetchLocation :: m Location

readEnv :: (MonadReader t m, Has a t) => m a
readEnv = reader getter

-- log :: (MonadReader t m, Has (Logger m) t, Show a) => a -> m ()
log s   = readEnv >>= (`runLogger` s)
-- sleep :: (MonadReader t m, Has (Sleeper m) t) => Microseconds -> m ()
sleep s = readEnv >>= (`runSleeper` s)

sleepWithDelay :: (HasSleeper r m, Has Seconds r, MonadReader r m) => m ()
sleepWithDelay = readEnv >>= (sleep . secsToMicrosecs)

get :: MonadRef m => Ref m AppData -> m AppData
get = readRef
put :: MonadRef m => Ref m AppData -> AppData -> m ()
put = writeRef
```

```haskell
-- Effectful program composing multiple effects / capabilities
type MonadAppCommon r m =
  (MonadReader r m, MonadRef m, MonadSpaceStation m, HasLogger r m, HasSleeper r m, Has Seconds r)

trackSpaceStation :: (MonadAppCommon r m, Has Count r, MonadCatch m) => m ()
trackSpaceStation = catch program logFailure
  where
    program :: (MonadAppCommon r m, Has Count r) => m ()
    program = do
      initialCount    <- readEnv
      initialLoc      <- fetchLocation
      initialRef      <- newRef (AppData initialLoc (initialCount - 1) [])
      finalSpeeds     <- trackSpeed initialRef <&> speeds
      let speedsString = show (reverse finalSpeeds)
      log $ "Successfully tracked ISS speed, result is: " ++ speedsString
    logFailure (e :: ISSException) =
      log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: MonadAppCommon r m => Ref m AppData -> m AppData
trackSpeed ref = do
  state <- get ref
  if remainingRequests state <= 0
    then log "No more requests to send to ISS" $> state
    else do
      sleepWithDelay
      currentLoc <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      put ref newState
      trackSpeed ref

-- Effect and app interpreters for semantics (deconstruction / elimination)
newtype App a = App { runApp :: Env App -> IO a }
  deriving (Functor, Applicative, Monad, MonadThrow, MonadCatch, MonadReader (Env App),
            MonadRef, MonadIO) via ReaderT (Env App) IO

instance (HasLogger r m, Has Request r, Monad m, MonadReader r m, MonadCatch m, MonadIO m)
           => MonadSpaceStation m where
  fetchLocation = do
    request <- readEnv
    body <- httpJSON request `catch` (throwM . ISSException)
    getResponseBody body `tapM` log

newtype Logger m     = Logger { runLogger :: forall a. Show a => a -> m () }
newtype Sleeper m    = Sleeper { runSleeper :: Microseconds -> m () }
data Env m           = Env { conf :: Config, logger :: Logger m, sleeper :: Sleeper m }
newtype ISSException = ISSException SomeException deriving Show
instance Exception ISSException

-- 'Has' boilerplate for granular environment access, can be auto-generated
instance MonadReader (Env m) m => Has (Logger m) (Env m) where getter  = reader logger
instance MonadReader (Env m) m => Has (Sleeper m) (Env m) where getter = reader sleeper
instance MonadReader (Env m) m => Has Seconds (Env m) where getter     = reader (requestDelay . conf)
instance MonadReader (Env m) m => Has Count (Env m) where getter       = reader (totalRequests . conf)
instance MonadReader (Env m) m => Has Request (Env m) where getter     = reader (httpRequest . conf)

type HasLogger r m  = Has (Logger m) r
type HasSleeper r m = Has (Sleeper m) r
```

A.6 Combining Mutable References, Tagless Final, the ReaderT Pattern and MonadBase

The code below contrives to illustrate a small tagless final DSL abstracting away the use of an `MVar` to support concurrent mutable state. This is not appropriate for our simple application, but serves as a concrete example combining `MonadBase`, `Has` and mutable references.

```haskell
main :: IO ()
main = let logger  = Logger (liftIO . print)
           sleeper = Sleeper (liftIO . threadDelay . getMicroseconds)
       in do
         emptyState <- newEmptyMVar
         runApp trackSpaceStation $ Env config logger sleeper emptyState

-- Full code omitted, it's mostly duplicated.
```

```haskell
data Env m = Env {
  conf :: Config, logger :: Logger m, sleeper :: Sleeper m, state :: MVar AppData
}
instance MonadReader (Env m) m => Has (MVar AppData) (Env m) where getter =
  reader state

class MonadSpaceStationState m where
  get :: m AppData
  put :: AppData -> m ()

newtype UninitialisedISSState = UninitialisedISSState String deriving Show
instance Exception UninitialisedISSState

instance (MonadBase IO m, MonadThrow m, MonadReader r m, Has (MVar AppData) r)
                                          => MonadSpaceStationState m where
  get = do
    mvar      <- readEnv
    maybeData <- tryReadMVar mvar
    let exc    = UninitialisedISSState "Read ISS state before it was initialised"
    maybe (throwM exc) return maybeData

  put newState = do
    mvar <- readEnv
    ifM (isEmptyMVar mvar) (putMVar mvar newState) (void $ swapMVar mvar newState)
```

We embed an `MVar` into our environment; this requires an effectful call to `newEmptyMVar` in `main`. This is projected out of the environment with a `Has` instance, used in the instance of `MonadSpaceStationState`. We introduce another custom exception, since it is possible for the reference to be unpopulated. Note also that operations `newEmptyMVar`, `tryReadMVar`, `isEmptyMVar`, `putMVar` and `swapMVar` are taken from the *lifted-base* library. This exports standard operations written in terms of the abstractions `MonadBase IO` and `MonadBaseControl IO` as opposed to `MonadIO` which has the base monad *hardcoded* to `IO`. We do not want to block until the variable is populated, hence the *try* function variant and monadic if expression—`ifM`.

A.7 Method Dictionary Embedding of fetchLocation

This demonstrates how we can use the method dictionary technique to embed potentially large dependencies in the environment. Syntax is provided to project this `fetchLoc` out of `Env`, leading to *identical function bodies* within `trackSpaceStation` and `trackSpeed`: only the class constraint grouping `MonadAppCommon` needed to change. This now looks more similar to the Scala example of §B.2 where we explicitly declare instances rather than letting the compiler resolve them for us. This induces an appealing pattern, and forms the basis of the *rio* framework; built by the author of the original ReaderT pattern blog post [76, 93].

```haskell
main :: IO ()
main = let logger       = Logger (liftIO . print)
           sleeper      = Sleeper (liftIO . threadDelay . getMicroseconds)
           spaceStation = SpaceStation fetchLocationImpl
       in runApp trackSpaceStation $ Env config logger sleeper spaceStation

-- Syntax for extracting the operation from the environment and running it
fetchLocation :: (MonadReader t m, Has (SpaceStation m) t) => m Location
fetchLocation = readEnv >>= runSpaceStation

-- Full code omitted. Identical except for the modified grouping below
type MonadAppCommon r m =
  (MonadReader r m, MonadRef m, HasLogger r m, HasSleeper r m,
   Has Seconds r, HasSpaceStation r m)

-- Embedding / projection of this dependency in the environment
newtype SpaceStation m = SpaceStation { runSpaceStation :: m Location }
data Env m             = Env { conf :: Config, ..., fetchLoc :: SpaceStation m }
instance MonadReader (Env m) m => Has (SpaceStation m) (Env m) where getter =
  reader fetchLoc

-- Convenient type alias to use in effectful programs
type HasSpaceStation r m = Has (SpaceStation m) r
```

A.8 Coproduct Framework

Coproduct framework used in the implementation of §7.1.2, §A.9, and A.10. We use the same idioms as the original paper [70]. The *free* library provides `Free` and the corresponding monad instance [133].

```haskell
module Coproduct where
```

```
import Control.Monad.Free (Free (Free))

data (:+:) f g a = InL (f a) | InR (g a) deriving Functor

class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a

instance Functor f => f :<: f where
  inj = id

instance (Functor f, Functor g) => f :<: (f :+: g) where
  inj = InL

instance {-# OVERLAPS #-} (Functor f, Functor g, Functor h, f :<: g) => f :<: (h :+: g) where
  inj = InR . inj

inject :: (f :<: g) => f (Free g a) -> Free g a
inject = Free . inj
```

A.9 Example Implementation: Free Monads

Uses a user-defined DSL for *all* effects; similar to the naive example of §5.2 using transformers for all effects. This makes it more easily comparable with our other examples. An improved version, mixing several of the techniques we have learned, is included in Appendix A.10.

```
module FreeMonads (main) where

import           Common                (AppData (..),
                                        Config (httpRequest, requestDelay, totalRequests),
                                        Location, Microseconds (getMicroseconds),
                                        config, secsToMicrosecs, tapM,
                                        updateState)
import           Control.Concurrent    (threadDelay)
import           Control.Monad.Catch   (MonadCatch, MonadThrow, SomeException)
import qualified Control.Monad.Catch   as MC (catch)
import           Control.Monad.Except  (MonadIO (..))
import           Control.Monad.Free    (Free (..), iterM)
import           Control.Monad.Reader  (MonadReader, ReaderT (..))
import qualified Control.Monad.Reader  as MR (ask, reader)
import           Coproduct             (inject, type (:+:) (..), type (:<:))
import           Data.Functor          ((<&>))
import           Network.HTTP.Simple   (getResponseBody, httpJSON)
import           Prelude               hiding (log)

main :: IO ()
main = let program = trackSpaceStation :: Free (Effects Config SomeException) ()
       in runApp (iterM eval program) config

-- Independent instructions combined into a DSL using coproducts
type Effects r e        = Ask r :+: (Sleep :+: (Log :+: (Catch e :+: FetchLocation)))
data Log f              = Log String f                    deriving Functor
data Sleep f            = Sleep Microseconds f            deriving Functor
newtype Ask r f         = Ask (r -> f)                    deriving Functor
newtype FetchLocation f = FetchLocation (Location -> f)   deriving Functor
data Catch e f          = Catch f (e -> f)                deriving Functor
data State s f          = Get (s -> f) | Put s f          deriving Functor

-- Effect syntax (construction / introduction)
log :: (Log :<: f) => String -> Free f ()
log s = inject $ Log s $ Pure ()

sleep :: (Sleep :<: f) => Microseconds -> Free f ()
sleep s = inject $ Sleep s $ Pure ()

ask :: (Ask r :<: f) => Free f r
ask = inject $ Ask Pure

reader :: (Ask r :<: f) => (r -> a) -> Free f a
reader = (ask <&>)

fetchLocation :: (FetchLocation :<: f) => Free f Location
fetchLocation = inject $ FetchLocation Pure

catch :: (Catch e :<: f) => Free f a -> (e -> Free f a) -> Free f a
catch program handle = inject $ Catch program handle
```

```haskell
get :: (State s :<: f) => Free f s
get = inject $ Get pure

put :: (State s :<: f) => s -> Free f ()
put s = (inject . Put s) $ pure ()

-- Effectful program composing multiple effects / capabilities
trackSpaceStation :: (Sleep :<: f, Log :<: f, FetchLocation :<: f,
                      Ask Config :<: f, Catch SomeException :<: f) => Free f ()
trackSpaceStation = catch program logFailure
  where
    program =  do
      initialCount <- reader totalRequests
      initialLoc   <- fetchLocation
      finalSpeeds  <- execState (AppData initialLoc (initialCount - 1) []) trackSpeed <&> speeds
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure (e :: SomeException) = log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: (Log :<: f, Sleep :<: f, FetchLocation :<: f, Ask Config :<: f)
                                          =>  Free (State AppData :+: f) ()
trackSpeed = do
  state <- get
  if remainingRequests state <= 0
    then log "No more requests to send to ISS"
    else do
      reader requestDelay >>= sleep . secsToMicrosecs
      currentLoc <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      put newState
      trackSpeed

-- Effect and app interpreters for semantics (deconstruction / elimination)
newtype App a = App { runApp :: Config -> IO a }
  deriving (Functor, Applicative, Monad, MonadIO,  MonadThrow, MonadCatch,
            MonadReader Config) via ReaderT Config IO

class Functor f => Eval f m where
  eval :: f (m a) -> m a

instance (Eval f m, Eval g m) => Eval (f :+: g) m where
  eval (InL f) = eval f
  eval (InR g) = eval g

instance MonadIO m => Eval Log m where
  eval (Log s a) = doLog s >> a

instance MonadIO m => Eval Sleep m where
  eval (Sleep ms a) = liftIO (threadDelay $ getMicroseconds ms) >> a

instance MonadReader r m => Eval (Ask r) m where
  eval (Ask useEnvironment) = MR.ask >>= useEnvironment

instance (MonadReader Config m, MonadIO m) => Eval FetchLocation m where
  eval (FetchLocation useLocation) = do
    body <- MR.reader httpRequest >>= httpJSON
    getResponseBody body `tapM` doLog >>= useLocation

instance MonadCatch m => Eval (Catch SomeException) m where
  eval (Catch program handle) = MC.catch program handle

doLog :: MonadIO m => forall a. Show a => a -> m ()
doLog = liftIO . print

execState :: Functor f => s -> Free (State s :+: f) a -> Free f s
execState s (Pure _)               = Pure s
execState s (Free (InL (Get k)))   = execState s (k s)
execState _ (Free (InL (Put s m))) = execState s m
execState s (Free (InR f))         = Free (execState s <$> f)
```

A.10 Example Implementation: Free Monads, Improved

An alternative version of §A.9 with several adjustments based on what we have learnt in Chapter 6. Enumerating the high-level changes:

1. The `Ask` instruction is leaking implementation details from the reader effect. This is merely semantics, as opposed to an intrinsic element of our domain. Instead, use domain-specific instructions with accompanying syntax used in the effectful program:

```
newtype GetDelay f        = GetDelay (Seconds -> f)        deriving Functor
newtype GetRequestCount f = GetRequestCount (Count -> f)   deriving Functor
```

2. In the *interpretations* of these new environment-based instructions, deploy the `Has` pattern described in §6.2 and used in §6.3.

```
instance (Has Seconds r, MonadReader r m) => Eval GetDelay m where
  eval (GetDelay useDelay) = readEnv >>= useDelay
```

   Logging is embedded in a environment function record using the method dictionary technique. This improves modularity by avoiding the need for `MonadIO` just to perform logging in type class instances.

3. Instead of a bespoke mini-DSL for state operations, use the `StateT` transformer. This demonstrates how we can combine the free monadic and MTL styles. However, we have to explicitly lift values of type `Free f` into `StateT`, which is undesirable. Using `MonadTrans`, we can write syntax for our DSLs that enable the emedding of instructions into any transformer:

```
sleep :: (Sleep :<: f, MonadTrans t) => Microseconds -> t (Free f) ()
sleep s = lift $ inject $ Sleep s $ Pure ()
```

   However, this is complicated and completely ruins type inference. Moreover, to do this for `catch` we stumble into the problem of higher-order effects described in §6.4.2. We would require `MonadTransControl` to lift `catch` through `StateT`. Therefore, this is not recommended; serving only to show that it is *possible*.

4. We use a more expressive custom exception `ISSException` as in §6.3.

Below is the full improved free monad implementation.

```
module FreeMonadsAlternate (main) where

import           Common                (AppData (..), Config (..), Count,
                                        Location, Microseconds (..), Seconds,
                                        config, secsToMicrosecs, tapM,
                                        updateState)
import           Control.Concurrent    (threadDelay)
import           Control.Monad.Catch   (Exception, MonadCatch, MonadThrow,
                                        SomeException)
import qualified Control.Monad.Catch   as MC (catch, throwM)
import           Control.Monad.Except  (MonadIO (..))
import           Control.Monad.Free    (Free (..), iterM)
import           Control.Monad.Reader  (MonadReader, ReaderT (..))
import qualified Control.Monad.Reader  as MR (reader)
import           Control.Monad.State   (StateT, execStateT, get, lift, put)
import           Coproduct             (inject, type (:+:) (..), type (:<:))
import           Data.Functor          (($>), (<&>))
import           Data.Has
import           Network.HTTP.Simple   (Request, getResponseBody, httpJSON)
import           Prelude               hiding (log)

-- Improved free monad implementation: granular environment, Has, StateT, custom exception
main :: IO ()
main = let program = trackSpaceStation :: Free (Effects Config ISSException) ()
       in runApp (iterM eval program) (Env config (Logger $ liftIO . print))

-- Independent instructions combined into a DSL using coproducts
type Effects r e          =
  GetDelay :+: (GetRequestCount :+: (Sleep :+: (Log :+: (Catch e :+: FetchLocation))))
data Log f                = Log String f                    deriving Functor
data Sleep f              = Sleep Microseconds f            deriving Functor
newtype GetDelay f        = GetDelay (Seconds -> f)         deriving Functor
newtype GetRequestCount f = GetRequestCount (Count -> f)    deriving Functor
newtype FetchLocation f   = FetchLocation (Location -> f)   deriving Functor
data Catch e f            = Catch f (e -> f)                deriving Functor

-- Effect syntax (construction / introduction)
log :: (Log :<: f) => String -> Free f ()
log s = inject $ Log s $ Pure ()

sleep :: (Sleep :<: f) => Microseconds -> Free f ()
```

```haskell
sleep s = inject $ Sleep s $ Pure ()

getRequestCount :: (GetRequestCount :<: f) => Free f Count
getRequestCount = inject $ GetRequestCount Pure

getDelay :: (GetDelay :<: f) => Free f Seconds
getDelay = inject $ GetDelay Pure

fetchLocation :: (FetchLocation :<: f) => Free f Location
fetchLocation = inject $ FetchLocation Pure

catch :: (Catch e :<: f) => Free f a -> (e -> Free f a) -> Free f a
catch program handle = inject (Catch program handle)

doLog :: (MonadReader r m, HasLogger r m, Show a) => a -> m ()
doLog a = readEnv >>= (`runLogger` a)

readEnv :: (MonadReader t m, Has a t) => m a
readEnv = MR.reader getter

-- Effectful program composing multiple effects / capabilities
trackSpaceStation :: (GetRequestCount :<: f, GetDelay :<: f, Sleep :<: f, Log :<: f,
                      FetchLocation :<: f, Catch ISSException :<: f) => Free f ()
trackSpaceStation = catch program logFailure
  where
    program =  do
      initialCount <- getRequestCount
      initialLoc   <- fetchLocation
      finalSpeeds  <- execStateT trackSpeed (AppData initialLoc (initialCount - 1) []) <&> speeds
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure (e :: ISSException) = log $ "Failed to call ISS with error: " ++ show e

trackSpeed :: (GetDelay :<: f, Log :<: f, Sleep :<: f, FetchLocation :<: f)
                                      => StateT AppData (Free f) AppData
trackSpeed = do
  state <- get
  if remainingRequests state <= 0
    then lift $ log "No more requests to send to ISS" $> state
    else do
      lift $ getDelay >>= sleep . secsToMicrosecs
      currentLoc <- lift fetchLocation
      let (speed, newState) = updateState currentLoc state
      lift $ log $ "Current speed is: " ++ show speed
      put newState
      trackSpeed

-- Effect and app interpreters for semantics (deconstruction / elimination)
newtype App a = App { runApp :: Env App -> IO a }
  deriving (Functor, Applicative, Monad, MonadIO,  MonadThrow, MonadCatch,
            MonadReader (Env App)) via ReaderT (Env App) IO
newtype Logger m     = Logger { runLogger :: forall a. Show a => a -> m () }
type HasLogger r m   = Has (Logger m) r
data Env m           = Env { conf :: Config, logger :: Logger m }
newtype ISSException = ISSException SomeException deriving Show
instance Exception ISSException

class Functor f => Eval f m where
  eval :: f (m a) -> m a

instance (Eval f m, Eval g m) => Eval (f :+: g) m where
  eval (InL f) = eval f
  eval (InR g) = eval g

instance (MonadReader r m, HasLogger r m) => Eval Log m where
  eval (Log s a) = doLog s >> a

instance (MonadIO m) => Eval Sleep m where
  eval (Sleep ms a) = liftIO (threadDelay $ getMicroseconds ms) >> a

instance (Has Seconds r, MonadReader r m) => Eval GetDelay m where
  eval (GetDelay useDelay) = readEnv >>= useDelay

instance (Has Count r, MonadReader r m) => Eval GetRequestCount m where
  eval (GetRequestCount useRequestCount) = readEnv >>= useRequestCount
```

```
instance (Has Request r, HasLogger r m, MonadReader r m, MonadCatch m,
          MonadIO m) => Eval FetchLocation m where
  eval (FetchLocation useLocation) = do
    request <- readEnv
    body    <- httpJSON request `MC.catch` (MC.throwM . ISSException)
    getResponseBody body `tapM` doLog >>= useLocation

instance (MonadCatch m) => Eval (Catch ISSException) m where
  eval (Catch program handle) = MC.catch program handle

instance MonadReader (Env m) m => Has Seconds (Env m) where getter    = MR.reader (requestDelay . conf)
instance MonadReader (Env m) m => Has Count (Env m) where getter      = MR.reader (totalRequests . conf)
instance MonadReader (Env m) m => Has Request (Env m) where getter    = MR.reader (httpRequest . conf)
instance MonadReader (Env m) m => Has (Logger m) (Env m) where getter = MR.reader logger
```

A.11 Example Implementation: Extensible Effects

We use the *polysemy* extensible effects library as our effect system [108]. This has gained popularity in the Haskell community and many systems built upon this framework have been deployed to production.

```
module ExtensibleEffects (main)  where

import           Common                 (AppData (..), Config (..), Location,
                                         Microseconds (..), config,
                                         secsToMicrosecs, tapM, updateState)
import           Control.Concurrent     (threadDelay)
import           Control.Monad.IO.Class (liftIO)
import           Data.Function          ((&))
import           Data.Functor           (void, (<&>))
import           GHC.Exception          (SomeException)
import           Network.HTTP.Simple    (getResponseBody, httpJSON)
import           Polysemy               (Embed, Member, Members, Sem, embed,
                                         interpret, runM)
import           Polysemy.Error
import qualified Polysemy.Internal      as P
import           Polysemy.Reader
import           Polysemy.State         (State, execState, get, put)
import           Polysemy.Trace         (Trace, trace, traceToStdout)
import           Prelude                hiding (log)

main :: IO ()
main = interpretISS trackSpaceStation

-- Independent instructions as GADTs
data Sleep m a          where Sleep :: Microseconds -> Sleep m ()
data FetchLocation m a where FetchLocation :: FetchLocation m Location

-- Effect syntax (construction / introduction)
log :: (Member Trace r, Show a)  => a -> Sem r ()
log = trace . show

sleep :: Member Sleep r => Microseconds -> Sem r ()
sleep = P.send . Sleep

sleepWithDelay :: Members '[Sleep, Reader Config] r => Sem r ()
sleepWithDelay = asks requestDelay >>= (sleep . secsToMicrosecs)

fetchLocation :: Member FetchLocation r => Sem r Location
fetchLocation = P.send FetchLocation

-- Effectful program composing multiple effects / capabilities
trackSpaceStation :: Members '[Sleep, Trace, FetchLocation, Reader Config,
                               Error ISSError] r => Sem r ()
trackSpaceStation = catch @ISSError program logFailure
  where
    program = do
      initialCount    <- asks totalRequests
      initialLoc      <- fetchLocation
      let initialState = AppData initialLoc (initialCount - 1) []
      finalSpeeds     <- execState initialState trackSpeed <&> speeds
      log $ "Successfully tracked ISS speed, result is: " ++ show (reverse finalSpeeds)
    logFailure e = log $ "Failed to call ISS with error: " ++ show e

trackSpeed ::  Members '[Sleep, Trace, FetchLocation, Reader Config,
                        State AppData] r => Sem r ()
```

```
trackSpeed = do
  state <- get
  if remainingRequests state <= 0
    then log "No more requests to send to ISS"
    else do
      sleepWithDelay
      currentLoc <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      put newState
      trackSpeed

-- Effect and app interpreters for semantics (deconstruction / elimination)
newtype ISSError = ISSError SomeException deriving Show

runFetchLocation :: (Members '[Embed IO, Reader Config, Trace, Error ISSError] r)
                                    => Sem (FetchLocation ': r) a -> Sem r a
runFetchLocation = interpret \case
  FetchLocation -> do
    request <- asks httpRequest
    body    <- fromExceptionVia ISSError (liftIO $ httpJSON request)
    getResponseBody body `tapM` log

sleepToIO :: Member (Embed IO) r => Sem (Sleep ': r) a -> Sem r a
sleepToIO = interpret \case Sleep ms -> embed (threadDelay $ getMicroseconds ms)

interpretISS :: Sem '[FetchLocation, Error ISSError, Sleep, Trace, Reader Config, Embed IO] a -> IO ()
interpretISS prog =
  runFetchLocation prog
    & runError @ISSError
    & sleepToIO
    & traceToStdout
    & runReader config
    & runM
    & void
```

A.12 Example Implementation: Extensible Effects, Improved

We improve upon the implementation of §A.11 using the proposals in §7.4.1. Specifically:

1. Use multiple instances of `Reader` for more granular environment passing.
2. Use `AtomicState`, interpreting it using the regular `State` interpreter as a proxy.
3. Push the error handling into an interpretation function which trades an `Error` effect for a `Log`.

```
module ExtensibleEffectsAlternate (main)  where

import           Common                (AppData (..), Config (..), Count,
                                        Location, Microseconds (..), Seconds,
                                        Speeds, config, secsToMicrosecs, tapM,
                                        updateState)
import           Control.Concurrent    (threadDelay)
import           Control.Monad.IO.Class (liftIO)
import           Data.Function         ((&))
import           Data.Functor          ((<&>))
import           GHC.Exception         (SomeException)
import           Network.HTTP.Simple   (Request, getResponseBody, httpJSON)
import           Polysemy              (Embed, Member, Members, Sem, embed,
                                        interpret, runM)
import           Polysemy.AtomicState  (AtomicState, atomicGet, atomicPut,
                                        execAtomicStateViaState)
import           Polysemy.Error        (Error, fromExceptionVia, runError)
import qualified Polysemy.Internal     as P
import           Polysemy.Reader
import           Polysemy.Trace        (Trace, trace, traceToStdout)
import           Prelude               hiding (log)

main :: IO ()
main = interpretISS trackSpaceStation

-- Independent instructions as GADTs
data Sleep m a          where Sleep :: Microseconds -> Sleep m ()
data FetchLocation m a where FetchLocation :: FetchLocation m Location
```

```haskell
-- Effect syntax (construction / introduction)
log :: (Member Trace r, Show a)  => a -> Sem r ()
log = trace . show

sleep :: Member Sleep r => Microseconds -> Sem r ()
sleep = P.send . Sleep

sleepWithDelay :: Members '[Sleep, Reader Seconds] r => Sem r ()
sleepWithDelay = ask >>= (sleep . secsToMicrosecs)

fetchLocation :: Member FetchLocation r => Sem r Location
fetchLocation = P.send FetchLocation

-- Effectful program composing multiple effects / capabilities
trackSpaceStation :: Members '[Sleep, Trace, FetchLocation, Reader Count, Reader Seconds,
                               Error ISSError] r => Sem r Speeds
trackSpaceStation = do
  initialCount    <- ask
  initialLoc      <- fetchLocation
  let initialState = AppData initialLoc (initialCount - 1) []
  execAtomicStateViaState initialState trackSpeed <&> reverse . speeds

trackSpeed ::  Members '[Sleep, Trace, FetchLocation, Reader Seconds, AtomicState AppData] r => Sem r ()
trackSpeed = do
  state <- atomicGet
  if remainingRequests state <= 0
    then log "No more requests to send to ISS"
    else do
      sleepWithDelay
      currentLoc <- fetchLocation
      let (speed, newState) = updateState currentLoc state
      log $ "Current speed is: " ++ show speed
      atomicPut newState
      trackSpeed

-- Effect and app interpreters for semantics (deconstruction / elimination)
newtype ISSError = ISSError SomeException deriving Show

runFetchLocation :: (Members '[Embed IO, Reader Request, Trace, Error ISSError] r) =>
                     Sem (FetchLocation ': r) a -> Sem r a
runFetchLocation = interpret \case
  FetchLocation -> do
    request <- ask
    body    <- fromExceptionVia ISSError (liftIO $ httpJSON request)
    getResponseBody body `tapM` log

errorToLog :: Member Trace r => Sem (Error ISSError ': r) Speeds -> Sem r ()
errorToLog prog = runError prog >>= \case
  Left (ISSError e) -> log $ "Failed to call ISS with error: " ++ show e
  Right finalSpeeds -> log $ "Successfully tracked ISS speed, result is: " ++ show finalSpeeds

sleepToIO :: Member (Embed IO) r => Sem (Sleep ': r) a -> Sem r a
sleepToIO = interpret \case Sleep ms -> embed (threadDelay $ getMicroseconds ms)

interpretISS :: Sem '[FetchLocation, Error ISSError, Sleep, Trace, Reader Count, Reader Request,
                      Reader Seconds, Embed IO] Speeds -> IO ()
interpretISS prog =
  runFetchLocation prog
    & errorToLog
    & sleepToIO
    & traceToStdout
    & runReader3 config totalRequests httpRequest requestDelay
    & runM

runReader3 :: i -> (i -> j) -> (i -> k) -> (i -> l) ->
               Sem (Reader j ': Reader k ': Reader l ': r) a -> Sem r a
runReader3 i f g h prog = runReader (h i) $ runReader (g i) $ runReader (f i) prog
```

B Scala Code

B.1 Model, Logic and Utility Functions

Named consistently with the Haskell examples.

```scala
package com.dantb.isstrackingscala.model

import cats.Show
import cats.syntax.all.*
import io.circe.Decoder
import io.circe.DecodingFailure
import org.http4s.Uri
import org.http4s.implicits.*

import scala.concurrent.duration.FiniteDuration

import concurrent.duration.DurationInt

final case class AppData(loc: Location, count: Int, speeds: List[Double]):
  def update(currentLoc: Location, speed: Double): AppData =
    AppData(currentLoc, count - 1, speed :: speeds)

final case class Config(issRequestUri: Uri, count: Int, delay: FiniteDuration)

final case class Location(lat: Double, lon: Double, timestamp: Long)

object Location:
  given Show[Location] = Show.fromToString
  given Decoder[Location] = cur =>
    def failure(msg: String): DecodingFailure = DecodingFailure(msg, Nil)
    def toDouble(key: String, value: String): Either[DecodingFailure, Double] =
      value.toDoubleOption.toRight(failure(s"Invalid Double inside String for key: $key"))
    val positionCur = cur.downField("iss_position")
    for
      lon       <- positionCur.get[String]("longitude").flatMap(toDouble("longitude", _))
      lat       <- positionCur.get[String]("latitude").flatMap(toDouble("latitude", _))
      timestamp <- cur.get[Long]("timestamp")
    yield Location(lat, lon, timestamp)

given Show[Throwable] = Show.show(e =>
  Option(e.getMessage())
    .orElse(Option(e.getCause()).flatMap(c => Option(c.getMessage())))
    .getOrElse(e.toString)
)

def config = Config(uri"http://api.open-notify.org/iss-now.json", 3, 5.seconds)

def computeSpeedKmPerHour(pointA: Location, pointB: Location): Double =
  val deltaMeters  = geoDistance(pointA, pointB)
  val deltaSeconds = pointA.timestamp - pointB.timestamp
  math.abs(deltaMeters / deltaSeconds) * 3.6

def geoDistance(pointA: Location, pointB: Location): Double =
  val Location(pointALat, pointALon, _) = pointA
  val Location(pointBLat, pointBLon, _) = pointB
  val deltaLat                          = math.toRadians(pointBLat - pointALat)
  val deltaLong                         = math.toRadians(pointBLon - pointALon)
  val a = math.pow(math.sin(deltaLat / 2), 2) + math.cos(math.toRadians(pointALat)) * math.cos(
    math.toRadians(pointBLat)
  ) * math.pow(math.sin(deltaLong / 2), 2)
  val greatCircleDistance = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
  val earthRadiusMeters   = 6371200
  earthRadiusMeters * greatCircleDistance
```

B.2 Example Implementation: Tagless Final in Scala

```scala
package com.dantb.isstrackingscala

import cats.MonadThrow
import cats.effect.*
import cats.syntax.all.*
import com.dantb.isstrackingscala.model.{*, given}
import org.http4s.*
import org.http4s.circe.CirceEntityDecoder.*
```

```
import org.http4s.client.Client
import org.http4s.ember.client.EmberClientBuilder

import scala.concurrent.duration.FiniteDuration

object Main extends IOApp.Simple:
  // Program entry point, Cats Effect runs the IO in the standard JVM 'main' method
  def run: IO[Unit] = EmberClientBuilder.default[IO].build.use { (client: Client[IO]) =>
    // Effect implementations providing semantics
    given Logger[IO]       = IO.println
    given Sleeper[IO]      = IO.sleep
    given SpaceStation[IO] = () => fetchLocationImpl(config.issRequestUri, client)
    trackSpaceStation(config)
  }

def fetchLocationImpl[F[_]: Logger: Concurrent](uri: Uri, client: Client[F]): F[Location] =
  client.expect[Location](Request(Method.GET, uri)).flatTap(log)

// Effect interfaces (tagless final DSLs)
trait Logger[F[_]]:
  def log(msg: String): F[Unit]
trait Sleeper[F[_]]:
  def sleep(duration: FiniteDuration): F[Unit]
trait SpaceStation[F[_]]:
  def fetchLocation(): F[Location]

// Convenient syntax
def log[F[_], A: Show](msg: A)(using l: Logger[F]): F[Unit]        = l.log(msg.show)
def sleep[F[_]](dur: FiniteDuration)(using s: Sleeper[F]): F[Unit] = s.sleep(dur)
def fetchLocation[F[_]]()(using ss: SpaceStation[F]): F[Location]  = ss.fetchLocation()

def trackSpaceStation[F[_]: MonadThrow: Logger: Sleeper: SpaceStation](config: Config): F[Unit] =
  def trackSpeed(data: AppData): F[AppData] =
    if data.count <= 0 then log("No more requests to send to ISS").as(data)
    else for
      _          <- sleep(config.delay)
      currentLoc <- fetchLocation()
      speed       = computeSpeedKmPerHour(data.loc, currentLoc)
      _          <- log(show"Current speed is: $speed")
      newState   <- trackSpeed(data.update(currentLoc, speed))
    yield newState

  val program: F[Unit] = for
    initialLoc <- fetchLocation()
    finalState <- trackSpeed(AppData(initialLoc, config.count - 1, Nil))
    _          <- log(show"Successfully tracked ISS speed, result is: ${finalState.speeds.reverse}")
  yield ()

  program.handleErrorWith(e => log(show"Failed to call ISS with error: $e"))
```

An equivalent function for the loop written tail recursively. However the interpreter used to instantiate `F[_]`, in this case the `IO` monad, must still be stack safe. Typically a technique called *trampolining* is used to facilitate this by using heap allocations rather than stack frames in the reified structure.

```
package com.dantb.isstrackingscala

import cats.Monad
import cats.syntax.all.*
import com.dantb.isstrackingscala.Main.*
import com.dantb.isstrackingscala.model.{*, given}

import scala.annotation.tailrec

import concurrent.duration.DurationInt

// Equivalent trackSpeed implementation which is tail recursive up to the effect monad
object TrackSpeedTailRec:
  def trackSpeed[F[_]: Monad: Logger: Sleeper: SpaceStation](data: AppData): F[AppData] =
    @tailrec
    def loop(remainingRequests: Int, acc: F[AppData]): F[AppData] =
      if remainingRequests <= 0 then acc <* log("No more requests to send to ISS")
      else
        val updateState: AppData => F[AppData] = newData =>
          for
            _          <- sleep(config.delay)
            currentLoc <- fetchLocation()
            speed       = computeSpeedKmPerHour(newData.loc, currentLoc)
            _          <- log(show"Current speed is: $speed")
          yield newData.update(currentLoc, speed)
```

```
      loop(remainingRequests - 1, acc >>= updateState)

  loop(data.count, data.pure[F])
```

B.3 Tagless Final Dependency Injection with Context Functions

B.3.1 Explanation Interleaved with Code We describe a novel pattern for dependency injection using context functions in Scala 3. We build up from first principles, highlighting the reasons for particular design decisions. This formulation is somewhat complex and pushes Scala's boundaries but, notably, it is *not* meta-programming. These are all standard language features. The construction at the end is reasonably terse and can be applied without a full understanding of the exact mechanics of the pattern. Given sufficient knowledge of the constituent standard language features, it is not a great leap to understand the pattern as a whole. The full code without commentary is included in Appendix B.3.2.

We can rewrite `trackSpaceStation` using context functions. With each variation we will increment the names and omit the function body since it is immaterial.

```
def trackSpaceStation2[F[_]](
  config: Config
): (MonadThrow[F], Logger[F], Sleeper[F], SpaceStation[F]) ?=> F[Unit]
```

The traits included in the return type are grouped using a comma. This produces the same compiled artefact. However, it is more verbose than context bounds and not reusable. Instead we could extract a type parameter to capture the grouping:

```
type Env[F[_], UsedIn] =
  (MonadThrow[F], Logger[F], Sleeper[F], SpaceStation[F]) ?=> UsedIn

def trackSpaceStation3[F[_]](config: Config): Env[F, F[Unit]]
```

This is now reusable, but the nesting of the return type is counter-intuitive. It is not clear that `F[Unit]` is the value returned. A more readable type would be preferable.

```
infix type Using[UsedIn, Eff[_]] = Eff[UsedIn]

type Env2[F[_]] = [UsedIn] =>>
  (MonadThrow[F], Logger[F], Sleeper[F], SpaceStation[F]) ?=> UsedIn

def trackSpaceStation4[F[_]](config: Config): F[Unit] Using Env2[F]
```

The *type lambda*, denoted by `=>>`, and *infix type*, named `Using` to be consistent with the `using` keyword, enable us to write this more intuitively. It can be read as "the environment `Env2[F]` is used to produce a value of type `F[Unit]`". This is reasonably terse, but we need the ability to *compose* groups of constraints, potentially defined in different places.

```
type Env3[F[_], UsedIn] = (MonadThrow[F], Logger[F]) ?=> UsedIn
type Env4[F[_], UsedIn] = (Sleeper[F], SpaceStation[F]) ?=> UsedIn
type Env5[F[_]]         = [UsedIn] =>> Env3[F, Env4[F, UsedIn]]

def trackSpaceStation5[F[_]](config: Config): F[Unit] Using Env5[F]
```

Here we compose two environments. Repeating `?=>` is not appealing and we are back to nesting; this time in the environment definition. We require two more definitions:

```
type Has[Eff[_[_]]] = [F[_], UsedIn] =>> Eff[F] ?=> UsedIn

infix type With[First[_[_], _], Second[_[_], _]] =
  [F[_], UsedIn] =>> First[F, Second[F, UsedIn]]
```

`Has` constitutes the *base case*. The "effect"—using the term loosely—`Eff` is used to produce a value of type `UsedIn`. Then, `With` allows us to *compose* two environments which have the correct shape. Specifically, this is the shape of a *tagless final DSL*: they must accept a type constructor—the `F`—and a regular type—the value produced by the function. `With` then nests these environments within each other, as we previously did manually. Now we may declare and compose more granular constraints:

```
type HasMonadThrow[F[_], UsedIn]       = Has[MonadThrow][F, UsedIn]
type HasLogger[F[_], UsedIn]           = Has[Logger][F, UsedIn]
type HasMonadThrowLogger[F[_], UsedIn] = With[HasMonadThrow, HasLogger][F, UsedIn]
type HasSleeper[F[_], UsedIn]          = Has[Sleeper][F, UsedIn]
type HasSpaceStation[F[_], UsedIn]     = Has[SpaceStation][F, UsedIn]

type Env6[F[_]] = [UsedIn] =>>
  With[With[HasMonadThrowLogger, HasSleeper], HasSpaceStation][F, UsedIn]

def trackSpaceStation6[F[_]](config: Config): F[Unit] Using Env6[F]
```

`HasMonadThrowLogger` composes `MonadThrow` with `Logger`, then `Env6` composes this with the rest of our dependencies. This is more modular. However it is *very* verbose due to the type applications and nesting. Scala unfortunately does not support partial type applications by default; we can use a trick to work around this:

```
infix type CompleteWith[First[_[_], _], Second[_[_], _]] =
  { type Use[F[_], UsedIn] = With[First, Second][F, UsedIn] }
```

This uses a *path-dependent type* to, in a sense, *delay* the application of the two types until later. We can make use of this in `Env7`:

```
type HasMonadThrowLoggerSleeper =
  Has[MonadThrow] With Has[Logger] CompleteWith Has[Sleeper]

type Env7[F[_]] = [UsedIn] =>>
  (HasMonadThrowLoggerSleeper#Use CompleteWith Has[SpaceStation])#Use[F, UsedIn]

def trackSpaceStation7[F[_]](config: Config): F[Unit] Using Env7[F]
```

Note that `HasMonadThrowLoggerSleeper` has no type parameters declared, making it more concise. We must now always *complete* a grouping with `CompleteWith` before the final `Has` constraint. Eventually we do need to apply the type to construct our environment; this is done with `#Use`. Symbolic type aliases could further reduce verbosity:

```
infix type *[First[_[_], _], Second[_[_], _]] =
  [F[_], UsedIn] =>> With[First, Second][F, UsedIn]

infix type ^[First[_[_], _], Second[_[_], _]] =
  CompleteWith[First, Second]

type Env8[F[_]] = [UsedIn] =>>
  (Has[MonadThrow] * Has[Logger] * Has[Sleeper] ^ Has[SpaceStation])#Use[F, UsedIn]

def trackSpaceStation8[F[_]](config: Config): F[Unit] Using Env8[F]
```

`&` would be preferable to `*` but we should not shadow the `&` used for *intersection types* in Scala. We chose `^` to loosely resemble a roof; the final piece of our environment grouping. `Env8` is not *too* much more verbose than our original `Env`, but it has one significant benefit: it supports the *composition* of multiple environments. We can now make use of this construction by reimplementing the ISS example:

```
type Common     = Has[Monad] * Has[Logger] * Has[Sleeper] ^ Has[SpaceStation]
type Env[F[_]] = [UsedIn] =>> Common#Use[F, UsedIn]

def trackSpaceStation[F[_]: ApplicativeThrow](config: Config): F[Unit] Using Env[F]
def trackSpeed[F[_]](data: AppData): F[AppData] Using Env[F]
```

We see that `trackSpeed` does not need the ability to throw or catch errors, so it does not need `MonadThrow`, only `Monad`. `trackSpaceStation` can then extend this common environment with a single `ApplicativeThrow` context bound to allow the use of `handleErrorWith`. This is a modularity improvement. It could be written entirely with context bounds—the benefit of this approach is that it encourages more granular functions by reducing the verbosity of type signatures. Combined with a linting tool to flag to the user that a particular type class constraint is unused, this can give rise to a powerful pattern for modular functional programming.

B.3.2 Full File with Modified Example The code in the explanation of §B.3.1 in a single file. Includes full function definitions for `trackSpaceStation` and `trackSpeed` using the same syntax as in §B.2.

```
package com.dantb.isstrackingscala

import cats.*
import cats.syntax.all.*
import com.dantb.isstrackingscala.model.{*, given}

// Rewriting the original function using a context function:
def trackSpaceStation2[F[_]](
  config: Config
): (MonadThrow[F], Logger[F], Sleeper[F], SpaceStation[F]) ?=> F[Unit] = trackSpaceStation(config)

// This is not reusable and no less verbose than context bounds. We extract a type:
type Env[F[_], UsedIn] =
  (MonadThrow[F], Logger[F], Sleeper[F], SpaceStation[F]) ?=> UsedIn
def trackSpaceStation3[F[_]](config: Config): Env[F, F[Unit]] = trackSpaceStation(config)

// The return type is not intuitive due to the nesting, can we make this more usable?
infix type Using[UsedIn, Eff[_]] = Eff[UsedIn]
type Env2[F[_]] = [UsedIn] =>>
```

```
  (MonadThrow[F], Logger[F], Sleeper[F], SpaceStation[F]) ?=> UsedIn

def trackSpaceStation4[F[_]](config: Config): F[Unit] Using Env2[F] = trackSpaceStation(config)

// This is reasonably terse. What if we want to group constraints in separate places?
type Env3[F[_], UsedIn] = (MonadThrow[F], Logger[F]) ?=> UsedIn
type Env4[F[_], UsedIn] = (Sleeper[F], SpaceStation[F]) ?=> UsedIn
type Env5[F[_]]         = [UsedIn] =>> Env3[F, Env4[F, UsedIn]]

def trackSpaceStation5[F[_]](config: Config): F[Unit] Using Env5[F] = trackSpaceStation(config)

// Fine, but we need to repeat ?=> and this requires nesting. Can we fix that?
type Has[Eff[_[_]]] = [F[_], UsedIn] =>> Eff[F] ?=> UsedIn
infix type With[First[_[_], _], Second[_[_], _]] =
  [F[_], UsedIn] =>> First[F, Second[F, UsedIn]]

type HasMonadThrow[F[_], UsedIn]       = Has[MonadThrow][F, UsedIn]
type HasLogger[F[_], UsedIn]           = Has[Logger][F, UsedIn]
type HasMonadThrowLogger[F[_], UsedIn] = With[HasMonadThrow, HasLogger][F, UsedIn]
type HasSleeper[F[_], UsedIn]          = Has[Sleeper][F, UsedIn]
type HasSpaceStation[F[_], UsedIn]     = Has[SpaceStation][F, UsedIn]

type Env6[F[_]] = [UsedIn] =>>
  With[With[HasMonadThrowLogger, HasSleeper], HasSpaceStation][F, UsedIn]

def trackSpaceStation6[F[_]](config: Config): F[Unit] Using Env6[F] = trackSpaceStation(config)

// This is more modular, but extremely verbose with all the type applications.
// Need to use dependent types to get around Scala's lack of partial type application:
infix type CompleteWith[First[_[_], _], Second[_[_], _]] =
  { type Use[F[_], UsedIn] = With[First, Second][F, UsedIn] }

type HasMonadThrowLoggerSleeper =
  Has[MonadThrow] With Has[Logger] CompleteWith Has[Sleeper]

type Env7[F[_]] = [UsedIn] =>>
  (HasMonadThrowLoggerSleeper#Use CompleteWith Has[SpaceStation])#Use[F, UsedIn]

def trackSpaceStation7[F[_]](config: Config): F[Unit] Using Env7[F] = trackSpaceStation(config)

// This is much more concise, though symbolic type aliases can improve it further:
infix type *[First[_[_], _], Second[_[_], _]] = [F[_], UsedIn] =>> With[First, Second][F, UsedIn]
infix type ^[First[_[_], _], Second[_[_], _]] = CompleteWith[First, Second]

type Env8[F[_]] = [UsedIn] =>>
  (Has[MonadThrow] * Has[Logger] * Has[Sleeper] ^ Has[SpaceStation])#Use[F, UsedIn]

def trackSpaceStation8[F[_]](config: Config): F[Unit] Using Env8[F] = trackSpaceStation(config)

// Altogether, this produces a more concise and modular implementation.
object FullModifiedExample:
  type Common    = Has[Monad] * Has[Logger] * Has[Sleeper] ^ Has[SpaceStation]
  type Env[F[_]] = [UsedIn] =>> Common#Use[F, UsedIn]

  def trackSpaceStation[F[_]: ApplicativeThrow](config: Config): F[Unit] Using Env[F] =
    val program = for
      initialLoc <- fetchLocation()
      finalState <- trackSpeed(AppData(initialLoc, config.count - 1, Nil))
      _          <- log(show"Successfully tracked ISS speed, result is: ${finalState.speeds.reverse}")
    yield ()
    program.handleErrorWith(e => log(show"Failed to call ISS with error: $e"))

  def trackSpeed[F[_]](data: AppData): F[AppData] Using Env[F] =
    if data.count <= 0 then log("No more requests to send to ISS").as(data)
    else for
      _          <- sleep(config.delay)
      currentLoc <- fetchLocation()
      speed       = computeSpeedKmPerHour(data.loc, currentLoc)
      _          <- log(show"Current speed is: $speed")
      newState   <- trackSpeed(data.update(currentLoc, speed))
    yield newState
```

B.4 Unit Testing with Transformer-based Interpreters

B.4.1 Explanation Interleaved with Code Programming with testing in mind, or even *starting* with tests using some form of *test-driven development*, tends to improve the structure of our programs. The process of writing unit tests can give a strong indication of the modularity and maintainability of our code. We illustrate how the composition of two monadic effects using transformers can give rise to a compelling pattern for white-box unit testing. Full test code is included in Appendix B.4.2.

We begin with the interpreter. The *cats* Scala library provides us with standard monads and transformers. We also define convenience functions for logging events and stubbing return values of our dependencies:

```
type EitherWriter[L, A]     = EitherT[[X] =>> Writer[L, X], Throwable, A]
type EitherWriterList[L, A] = EitherWriter[List[L], A]

def logInEitherT[F[_], L: Monoid, A](l: L, a: A): EitherWriter[L, A] =
  EitherT.liftF(Writer.tell(l)).as(a)
def logListInEitherT[F[_], L, A](l: L, a: A): EitherWriterList[L, A] =
  logInEitherT(List(l), a)
```

This composes `Writer` (the writing part of `State`) with `Either` in a *state-preserving* manner; as we have discussed previously in §3.1.2 and §6.4.1. We use `Writer` to *log* interactions using the `tell` function. `Either` is used to test failure. `EitherT` has the same shape as `MaybeT` defined in §5.1.

We want to test `trackSpaceStation`. What interactions would we like to assert against? Logging is not generally considered functionally significant, so we can stub out a `Logger[F]` instance which does nothing. We should test that the ISS abstraction `SpaceStation` is called when we expect it to be. The semantic blocking of `Sleeper` is also test-worthy, and has input we can capture. We can create a small test sum type to capture these possible "events":

```
enum TestEvent:
  case FetchedLocation
  case SleptFor(time: FiniteDuration)

type F[A] = EitherWriterList[TestEvent, A]

given Logger[F] = _ => ().pure[F]
```

Our test interpreter becomes `EitherWriterList[TestEvent, A]`, with a type alias for conciseness. This communicates that `F` may log values of type `TestEvent` within a `List` and can fail with a `Throwable`. Since `Writer` has a `Monad` instance, `EitherT` has a `MonadThrow` instance—just like `IO`. Hence it can instantiate our abstract `F[_]` in `trackSpaceStation` as long as we provide fake instances of our tagless final DSLs. We can avoid using an unnecessarily powerful concrete monad such as `cats.effect.IO`. Consider the happy path:

```
property("trackSpaceStation calls the ISS API 'count' times when 'count > 0'") {
  given Arbitrary[Config] = Arbitrary(genConfig(1, 100))

  forAll { (location: Location, config: Config) =>

    val singleLoopEvents  = List(SleptFor(config.delay), FetchedLocation)
    val allLoopEvents     = List.fill(config.count - 1)(singleLoopEvents).flatten
    val expectedEvents    = FetchedLocation :: allLoopEvents
    given SpaceStation[F] = () => logListInEitherT(FetchedLocation, location)
    given Sleeper[F]      = time => logListInEitherT(SleptFor(time), ())

    val result: F[Unit] = trackSpaceStation[F](config)

    assertEquals(result.value.run, (expectedEvents, Right(())))
  }
}
```

Note that we are using *property-based testing* to auto-generate data so we need not hand-roll it for every test. This style of testing can also reveal subtle bugs by exposing assumptions we did not realise we made. Humans are not good at randomness, thus our "random" choices of data and test cases can be deficient.

This example is the happy path: if our request count is 1 or more we expect the program to first fetch the initial location, then perform two interactions for each subsequent loop. This is captured in `expectedEvents`. The stubs of our DSLs log an event and stub the return value with arbitrary data. In this case, it is opaque what that data is used for: indeed, we should separately test the speed calculation logic. Our function under test is only useful for its effects, which why it returns an `F[Unit]`. In our assertion we *run* these effects by unwrapping our transformer and writer to produce a value of type `(List[TestEvent], Either[Throwable, Unit])`. We assert this against `(expectedEvents, Right(()))`: the list is as expected and the function succeeded with a `Right`. What about if the ISS API throws an exception?

```
property("trackSpaceStation handles any errors thrown by the ISS API") {
  given Arbitrary[Config]    = Arbitrary(genConfig(1, 100))
  given Arbitrary[Throwable] =
    Arbitrary(Gen.asciiPrintableStr.map(e => new Throwable(e)))

  forAll { (config: Config, error: Throwable) =>
    val expectedEvents    = List(FetchedLocation)
    given SpaceStation[F] = () =>
      EitherT(Writer.tell(List(FetchedLocation)).as(Left(error)))
    given Sleeper[F]      = time => logListInEitherT(SleptFor(time), ())

    val result = trackSpaceStation[F](config)

    assertEquals(result.value.run, (expectedEvents, Right(())))
  }
}
```

We do not need to actually *throw* an exception, just populate a `Left` with some throwable. This is all referentially transparent. The assertion is identical since our program recovers from all exceptions. This differs from the Haskell example of §A.5 which wraps failures in a custom exception; though that adjustment is simple to make by extending `Throwable` with a custom data type. We expect failure to happen after the first call to `fetchLocation`, so we only expect one `FetchedLocation` event in our list.

This is a reasonably terse approach for testing in a pure functional manner. It exposes some of the limitations in our program. For example, that configuring a *request count* does not reflect our intention. It would be better to configure a *speed count*; the number of requests is really just an implementation detail. Generating these config values makes this obvious. A huge count causes a stack overflow without using the stack safe version included in §B.2. This is something which would not be caught with regular hand-rolled data. Moreover, we should really make illegal states unrepresentable by using a stronger type than `Int`. Zero and negative numbers are not valid: this parsing could occur on program startup. A final point is that testing makes clear which elements of a function are redundant. In this case `issRequestUri` of `Config` is not used: we could make the function signature more granular.

We see that powerful tests can lead to more robust and concise programs. Improving the usability of testing is important to encourage more comprehensive coverage; though coverage by itself is not a good measure of test efficacy. Transformers can help in this regard.

As an aside, to demonstrate the flexibility of transformers, consider composing these two effects in the opposite order. If we do not need to preserve state, we can nest `Writer` inside `Either` using `WriterT`:

```
type WriterEitherList[L, A] =
  WriterT[[X] =>> Either[Throwable, X], List[L], A]

def tellValue[F[_]: Applicative, L: Monoid, A](l: L, a: A): WriterT[F, L, A]    =
  WriterT.tell(l).as(a)
def tellListValue[F[_]: Applicative, L, A](l: L, a: A): WriterT[F, List[L], A] =
  tellValue(List(l), a)
```

B.4.2 Full File with Tests

```
package com.dantb.isstrackingscala

import cats.*
import cats.data.*
import cats.syntax.all.*
import com.dantb.isstrackingscala.model.Config
import com.dantb.isstrackingscala.model.Location
import munit.ScalaCheckSuite
import org.http4s.Uri
import org.scalacheck.Arbitrary.arbitrary
import org.scalacheck.Prop.*
import org.scalacheck.*

import scala.concurrent.duration.FiniteDuration

class SpaceStationSpec extends ScalaCheckSuite:

  // Transformer-based testing interpreter, written generically
  type EitherWriter[L, A]     = EitherT[[X] =>> Writer[L, X], Throwable, A]
  type EitherWriterList[L, A] = EitherWriter[List[L], A]

  def logInEitherT[F[_], L: Monoid, A](l: L, a: A): EitherWriter[L, A]     =
    EitherT.liftF(Writer.tell(l)).as(a)
  def logListInEitherT[F[_], L, A](l: L, a: A): EitherWriterList[L, A]     =
```

```
  logInEitherT(List(l), a)

type WriterEitherList[L, A] = WriterT[[X] =>> Either[Throwable, X], List[L], A]

def tellValue[F[_]: Applicative, L: Monoid, A](l: L, a: A): WriterT[F, L, A]    =
  WriterT.tell(l).as(a)
def tellListValue[F[_]: Applicative, L, A](l: L, a: A): WriterT[F, List[L], A] =
  tellValue(List(l), a)

// Generators of arbitrary data, typically live elsewhere and can generally be auto-derived
def genLocation: Gen[Location] =
  for
    lat       <- arbitrary[Double]
    lon       <- arbitrary[Double]
    timestamp <- arbitrary[Long]
  yield Location(lat, lon, timestamp)

def genConfig(minCount: Int, maxCount: Int): Gen[Config] =
  for
    uri   <- Gen.const(Uri.unsafeFromString("notused.com"))
    count <- Gen.chooseNum(minCount, maxCount)
    delay <- arbitrary[FiniteDuration]
  yield Config(uri, count, delay)

given Arbitrary[Location] = Arbitrary(genLocation)

enum TestEvent:
  case FetchedLocation
  case SleptFor(time: FiniteDuration)

type F[A] = EitherWriterList[TestEvent, A]

given Logger[F] = _ => ().pure[F]

import TestEvent.*

property("trackSpaceStation calls the ISS API 'count' times when 'count > 0'") {
  given Arbitrary[Config] = Arbitrary(genConfig(1, 100))
  forAll { (location: Location, config: Config) =>
    val eventPairs        =
      List.fill(config.count - 1)(List(SleptFor(config.delay), FetchedLocation)).flatten
    val expectedEvents    = FetchedLocation :: eventPairs
    given SpaceStation[F] = () => logListInEitherT(FetchedLocation, location)
    given Sleeper[F]      = time => logListInEitherT(SleptFor(time), ())

    val result = trackSpaceStation[F](config)

    assertEquals(result.value.run, (expectedEvents, Right(())))
  }
}

property("trackSpaceStation calls the ISS API once when 'count <= 0'") {
  given Arbitrary[Config] = Arbitrary(genConfig(0, 0))
  forAll { (location: Location, config: Config) =>
    val expectedEvents    = List(FetchedLocation)
    given SpaceStation[F] = () => logListInEitherT(FetchedLocation, location)
    given Sleeper[F]      = time => logListInEitherT(SleptFor(time), ())

    val result = trackSpaceStation[F](config)

    assertEquals(result.value.run, (expectedEvents, Right(())))
  }
}

property("trackSpaceStation handles any errors thrown by the ISS API") {
  given Arbitrary[Config]    = Arbitrary(genConfig(2, 2))
  given Arbitrary[Throwable] = Arbitrary(Gen.asciiPrintableStr.map(e => new Throwable(e)))
  forAll { (config: Config, error: Throwable) =>
    val expectedEvents    = List(FetchedLocation)
    given SpaceStation[F] = () => EitherT(Writer.tell(List(FetchedLocation)).as(Left(error)))
    given Sleeper[F]      = time => logListInEitherT(SleptFor(time), ())

    val result = trackSpaceStation[F](config)

    assertEquals(result.value.run, (expectedEvents, Right(())))
```

```
    }
}
```

C Extended Explanations

C.1 The n^2 Instances Problem by Example

Consider the problem for a specific transformer, interface pair: `RealTimeT` and `MonadTime`. We can break this down into two parts. Firstly, `MonadTime` must be propagated up the stack through every transformer *above* `RealTimeT`. Secondly, the transformer needs to allow every interface *below* it in the stack to pass through it.

The first direction suggests we want an instance of `MonadTime` for arbitrary transformers:

```
instance (MonadTime m, MonadTrans t, Monad (t m)) => MonadTime (t m) where ...
```

This is a terse way of expressing the following. Given the prerequisites that:

1. The monad `m` supports the `MonadTime` operations.
2. There exists a way to produce a `t m` from an `m` (via `MonadTrans`).
3. The type `t m` has a monad.

Then, the type `t m` also supports the `MonadTime` operations. In practice, the `MonadTrans` instance should be provided alongside the definition of the transformer. The library in which `MonadTime` resides, *mock-time*, defines the above instance for arbitrary transformers using an `{-## OVERLAPPABLE ##-}` language pragma to allow overlapping instances. This is somewhat controversial, since type classes should generally be *coherent*: there should only be one valid instance for any type and type class pair[1].

An undisputable issue is that not all interfaces do support composition via `MonadTrans`, including some of our essential effects. This stems from a fundamental limitation of `lift` as a mechanism for composing monadic effects: it is not powerful enough to implement certain *higher-order effects*. These are operations which operate on other effects, such as `catchError` of `MonadError` and `local` of `MonadReader`. This is a more subtle issue requiring complex solutions and will be discussed once the more urgent problems are resolved in Chapter 6.

In general, we are reliant on transformers implementing `MonadTrans`. For example, the identity monad transformer `IdentityT` is used in the definition of `SpaceStationT` *purely* so that `MonadTrans` can be auto-derived. This aids extensibility by permitting different orderings with `RealTimeT` and any future transformers supporting composition via `MonadTrans`. This is yet more incidental complexity. Fortunately, for this particular ordering no extra boilerplate is required to propagate `threadDelay` *up* the stack. There is only one level to jump up to `App`.

We are less fortunate for the second direction: allowing other interfaces to pass through `RealTimeT`. It was necessary to declare an instance `MonadTrans RealTimeT` because the library does not provide one[2]. This typifies one of the problems with MTL: we must trust third parties to implement all the required components. Otherwise, prepare for some painful troubleshooting of issues that are completely irrelevant to the problem domain.

Luckily, instances of the standard effects *are* defined along with the library definition of `RealTimeT`. However, further boilerplate instances are required to lift `MonadLogger` and `MonadSpaceStation` through `RealTimeT`. This process raises more concerns. Troubleshooting the issue required knowledge of the modules in which transformers reside in order to pinpoint the problematic interface transformer pair. Modularity suffers as a result.

Moreover, `MonadLogger` defines a *default implementation* of its single operation for any `MonadTrans`, since `lift` is powerful enough for this simple effect. This is another method of auto-derivation turned on by a compiler setting. We obviate the overlapping instances issue but are still required to declare an unimplemented instance `MonadLogger (RealTimeT m)`.

Both of these are examples of *orphan instances*: they are not in the module of either the type class, such as `MonadLogger`, or the concrete type constructor, `RealTimeT`. Orphan instances are generally discouraged and produce compiler warnings. In combination with the coherence issue already mentioned, they make programs nonmodular. Changes to dependencies, such as new instance declarations, may cause compilation failure, or worse, *changes to runtime behaviour* [137]. To preclude this, we could add a `newtype` wrapper around `RealTimeT` and restart the process of derivation for every class in the system, producing yet more boilerplate.

C.2 Automating Derviation

We illustrate two techniques for automating derivation distinct from simply defining a manual instance of `MonadSpaceStation`.

We can create a *default implementation* of `fetchLocation` that works for any type implementing `MonadTrans`:

[1] Technically non-overlapping instances are a necessary but not sufficient condition for coherence [78].

[2] Its source code claims the omission is due to an auto-derivation failure, further illustrating MTL's limitations.

```
class Monad m => MonadSpaceStation m where
  fetchLocation :: m Location
  default fetchLocation ::
    (MonadSpaceStation m', MonadTrans t, m ~ t m') => m Location
  fetchLocation = lift fetchLocation
```

Here we need to give the compiler evidence that the the abstract type t is the same as m with `m ~ t m'`.

Alternatively, we could define an instance derivable by arbitrary transformers:

```
instance (MonadSpaceStation m, MonadTrans t, Monad (t m)) => MonadSpaceStation (t m) where
  fetchLocation = lift fetchLocation
```

This requires overlapping instances, as discussed in §5.3.1.

C.3 DSL Embeddings: Initial Tagged, Initial Tagless and Final Tagless

The following code originated within a single Haskell file, included in §C.3.4. We interleave commentary and example evaluations throughout the different DSL embeddings. These extensions are required:

```
{-# LANGUAGE FlexibleInstances         #-}
{-# LANGUAGE GADTs                     #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

module DSLEmbeddings where
```

The only notable extension is `NoMonomorphismRestriction` which is necessary to allow type inference in our tagless final example expressions. Kiselyov explains this in more depth [89].

C.3.1 Initial Tagged Embedding The most basic form of Hutton's razor [88] uses a simple union type as the instruction set. This is a rudimentary arithmetic DSL with integer literals and addition. Note that this is a *recursive data type*: leaf nodes in the union refer to the union itself. We call this an *object language* embedded within the host, or *metalanguage*—Haskell.

```
data BasicExp = BasicI Int | BasicAdd BasicExp BasicExp

evalBasic :: BasicExp -> Int
evalBasic (BasicI i)     = i
evalBasic (BasicAdd a b) = evalBasic a + evalBasic b
```

The object language is interpreted into the metalanguage with `evalBasic`. This reduces addition of object terms down to addition of values of type `Int` in Haskell.

We extend the basic DSL with instructions for *if-else* expressions and equality. This is not a typical modification. We do this because it is preferable to construct our narrative around a single example. There are two orthogonal problems to demonstrate. *Tagging* is an artefact of the lack of type-safety in our DSL. *Extension*—characterised in the expression problem [60]—constitutes adding an instruction to the DSL and observing its limitations.

```
data Exp = I Int
  | Add Exp Exp
  | Eq Exp Exp
  | IfElse Exp Exp Exp
```

```
-- Valid expressions
ex1 = Add (I 3) (I 4)
ex2 = Add ex1 (I 5)
ex3 = Eq (I 2) (I 7)
ex4 = IfElse ex3 ex1 (I 5)
-- Invalid expressions
ex5 = IfElse (I 1) ex1 ex2
ex6 = IfElse ex3 ex1 ex3
```

Examples `ex1`–`ex4` constitute *valid* expressions; adding and comparing integer literals. These gradually build up more complex expressions from primitive constructors. Unfortunately, it is also possible to compile expressions that are *invalid*: `ex5` and `ex6`. `ex5` uses an integer literal as the discriminator when it should be a *boolean-like* expression, whereas `ex6` returns expressions of different "types". These vagaries are symptomatic of the underlying problem: our DSL is not well-typed, detracting from its expressiveness. To remedy the issue, we need a `Result` ADT to enumerate the valid result types. This has ramifications when we try to interpret these expressions. Consider first an interpreter to *evaluate* the object language into the metalanguage:

```
data Result = RI Int | RB Bool deriving Show

eval :: Exp -> Result
eval (I i)             = RI i
eval (Add a b)         = case (eval a, eval b) of
  (RI a', RI b')       -> RI $ a' + b'
  (_, _)               -> error "1. Cannot add non-integers"
eval (Eq a b)          = case (eval a, eval b) of
  (RI a', RI b')       -> RB $ a' == b'
  (_, _)               -> error "2. Forbid equating expressions involving booleans"
eval (IfElse disc a b) = case (eval disc, eval a, eval b) of
  (RB d, RI a', RI b') -> if d then RI a' else RI b'
  (RB d, RB a', RB b') -> if d then RB a' else RB b'
  (RB _, _, _)         -> error "3. Returned expressions must have the same type"
  (_, _, _)            -> error "4. Discriminator must be a boolean"
```

We can no longer write an interpreter with the type of `evalBasic` for two reasons. It is now possible for evaluation to fail; that is, the function must be *partial*. Therefore, to preserve purity we would have to consider returning a `Maybe` to represent failure. The alternative is to ensure that expressions are validated before being evaluated. Regrettably, that any validation has taken place is opaque in the type signature. In the spirit of favouring parsing over validation, this is unsatisfactory [90].

Secondly, our target value could now be a `Bool` *or* an `Int`. This forces us to use `Result` to *untag* types in the case discriminator, and then *tag* in the return values. There are four branches of the pattern match that require tagging; we give the reason in error messages.

Note that we *can* define a *total* pretty printing evaluator which cannot fail:

```
view :: Exp -> String
view (I i) = show i
view (Add a b) = "(" ++ view a ++ " + " ++ view b ++ ")"
view (Eq a b) = "(" ++ view a ++ " == " ++ view b ++ ")"
view (IfElse disc a b) = "if " ++ view disc ++ " then " ++ view a ++ " else " ++ view b
```

This is possible because the target type `String` is static. The snag is that this interpreter would willingly print invalid expressions. To demonstrate:

```
*DSLEmbeddings> eval ex1
RI 7
*DSLEmbeddings> eval ex3
RB False
*DSLEmbeddings> eval ex6
*** Exception: 3. Returned expressions must have the same type
CallStack (from HasCallStack):
  error, called at DSLEmbeddings.hs:40:27 in main:DSLEmbeddings
*DSLEmbeddings> view ex4
"if (2 == 7) then (3 + 4) else 5"
*DSLEmbeddings> view ex5
"if 1 then (3 + 4) else ((3 + 4) + 5)"
```

Observe that `eval ex6` throws an error while `view ex5` happily prints an invalid expression.

C.3.2 Initial Tagless Embedding These problems of tagging and partial evaluation can be alleviated by incorporating better type-safety in the DSL. One means of achieving this in languages that support them is by using a *generalised algebraic data type*. `ExpIT` contains consistently-named data constructors:

```
data ExpIT a where
  I'      :: Int -> ExpIT Int
  Add'    :: ExpIT Int -> ExpIT Int -> ExpIT Int
  Eq'     :: ExpIT Int -> ExpIT Int -> ExpIT Bool
  IfElse' :: ExpIT Bool -> ExpIT b -> ExpIT b -> ExpIT b

-- Valid expressions
ex7 = Add' (I' 4) (I' 5)
ex8 = Eq' (I' 4) (I' 5)
ex9 = IfElse' ex8 ex7 (I' 6)
-- Invalid expressions no longer compile, a win!
-- ex10 = Add' (B' True) (I' 4)
-- ex11 = IfElse' ex8 (B' True) (I' 2)
```

It is now possible to assert our three object language invariants at compile time: addition is restricted to integers, the discriminator must be a `Bool` and the return value of an *if-else* expression must be constant. The polymorphic type parameter `a`, the characteristic feature of a *generalised* ADT, permits this. Now both our interpreters are *total and well-typed*:

```
evalIT :: Eq a => ExpIT a -> a
evalIT (I' i)             = i
evalIT (Add' a b)         = evalIT a + evalIT b
evalIT (Eq' a b)          = evalIT a == evalIT b
evalIT (IfElse' disc a b) = if evalIT disc then evalIT a else evalIT b

viewIT :: ExpIT a -> String
viewIT (I' i)             = show i
viewIT (Add' a b)         = "(" ++ viewIT a ++ " + " ++ viewIT b ++ ")"
viewIT (Eq' a b)          = "(" ++ viewIT a ++ " == " ++ viewIT b ++ ")"
viewIT (IfElse' disc a b) = "if " ++ viewIT disc ++ " then " ++ viewIT a ++ " else " ++ viewIT b
```

Note that `Eq` here is the standard Haskell type class for equality; an element of the metalanguage. GADTs are an appealing tool for typed eDSLs and they form the basis of extensible effect systems [22, 23]. Another advantage of this approach is that it is more clear which types are intrinsic to the DSL and which can be abstracted away since they are arbitrary. It is a completely *context free grammar*. Again we run the interpretations, noting that invalid expressions are now simply unrepresentable:

```
*DSLEmbeddings> evalIT ex9
6
*DSLEmbeddings> viewIT ex9
"if (4 == 5) then (4 + 5) else 6"
```

However, `ExpIT` suffers from a different problem: it lacks extensibility. Consider attempting to extend our DSL with integer multiplication:

```
data ExpIT a where
  --- other constructors...
  Mult :: ExpIT Int -> ExpIT Int -> ExpIT Int
```

This adds an instruction—a *row* in Wadler's characterisation [60]—to our instruction set; forcing changes to all existing interpreters to satisfy exhaustive pattern matching. Otherwise we get a match exception:

```
*DSLEmbeddings> viewIT $ Mult (I' 3) (I' 4)
"*** Exception: DSLEmbeddings.hs:(66,1)-(69,96): Non-exhaustive patterns in function viewIT
```

This is undesirable and reflects that this construction violates the expression problem.

C.3.3 Final Tagless Embedding Using a different tagless style we can solve the problems of *tagging*, *type-safety* and *extensibility* in one fell swoop. A *final embedding* is characterised by a parameterisation of the DSL over the interpretation. Instead of a data type, we declare classes to represent the two DSLs:

```
class AddFT a where
  int :: Int -> a
  add :: a -> a -> a

class EqFT a where
  eq :: Int -> Int -> a
  ifElse :: Bool -> a -> a -> a
```

The type parameter `a` is polymorphic *directly* over the metalanguage interpretation type. This is no longer context free: denotations of object terms are defined directly, though abstractly, in terms of host language types. Thus, the static guarantees of our DSL are inherited from the host language. This becomes clearer by seeing examples of interpreters:

```
instance AddFT Int where
  int = id
  add a b = a + b

instance AddFT String where
  int = show
  add a b = "(" ++ a ++ " + " ++ b ++ ")"

instance EqFT Bool where
  eq = (==)
  ifElse disc a b = if disc then a else b

instance EqFT String where
  eq a b = "(" ++ show a ++ " == " ++ show b ++ ")"
  ifElse disc a b = "if " ++ show disc ++ " then " ++ a ++ " else " ++ b
```

Interpreters in this embedding take the form of *type class instances*. Note that type classes are merely one—particularly useful—representation of a final DSL in a host language; an alternative approach involving OCaml *modules* was described in the inceptive paper of Carette et al. [91]. We provide interpretations of `AddFT` using `Int` and `String` in the metalanguage, and of `EqFT` using `Bool` and `String`. These DSLs are well-typed, just like GADTS, because they essentially defer to the type system of the metalanguage—Haskell. Now we can use these class operations as language terms, writing expressions without assigning any semantics:

```
-- ex12 :: AddFT a => a
ex12 = add (int 3) (int 5)
ex13 = add ex12 (int 2)
ex14 = eq ex12 ex13
ex15 = ifElse ex14 ex12 ex13
```

These examples are inferred as having polymorphic types, as the commented-out `ex12` signature illustrates. This type reads as: any `a` in which integers can be embedded, also supporting some notion of addition. Making this type monomorphic happens during interpretation where the type is inferred based on the *typed context*. In the REPL:

```
*DSLEmbeddings> ex12 :: Int
8
*DSLEmbeddings> ex15 :: String
"if False then (3 + 5) else ((3 + 5) + 2)"
```

Here we see one of the characteristics of a final embedding: the expressions are *reduced* to their representation in the host language. This precludes opportunities for transformation in the object language, such as optimisations based on

known reduction rules within our domain. Initial embeddings have been described as *context free grammars*, since they constitute a set of instructions free from any semantic domain [89]. It is possible to regain this lost power by interpreting a final embedding *back into an initial embedding*. This has applications in program optimisation, static analysis and testing [92].

Notably, we can now extend the DSL with multiplication without demanding recompilation of existing interpreters. We simply define a new type class and its instances:

```
class MultFT a where
  mult :: a -> a -> a

instance MultFT Int where
  mult = (*)

instance MultFT String where
  mult a b = "(" ++ a ++ " * " ++ b ++ ")"
```

```
ex16 = add (mult (int 4) (int 2)) (int 1)
ex17 = mult (int 3) (int 3)
ex18 = ifElse (eq ex16 ex17) (int 50) (int 51)
```

Some closing examples involving every instruction in our tagless final eDSLs:

```
*DSLEmbeddings> ex16 :: Int
9
*DSLEmbeddings> eq ex17 ex16 :: Bool
True
*DSLEmbeddings> ex18 :: String
"if True then 50 else 51"
*DSLEmbeddings> ex18 :: Bool

<interactive>:66:1: error:
    • No instance for (AddFT Bool) arising from a use of ‘ex18’
    • In the expression: ex18 :: Bool
      In an equation for ‘it’: it = ex18 :: Bool
```

The final example emphasises that the validity of a tagless final expression, potentially aggregating multiple DSLs, is determined by the typed context. `ex18` has two inferred type constraints: `(EqFT a, AddFT a)`. Thus, we may only interpret the expression in types which have interpretations—type class instances—for *both* constraints. In this case, `String` is the only type satisfying these requirements.

C.3.4 Full File with DSL Embeddings

```
{-# LANGUAGE FlexibleInstances         #-}
{-# LANGUAGE GADTs                     #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

module DSLEmbeddings where

data BasicExp = BasicI Int | BasicAdd BasicExp BasicExp

evalBasic :: BasicExp -> Int
evalBasic (BasicI i)     = i
evalBasic (BasicAdd a b) = evalBasic a + evalBasic b

data Exp = I Int
  | Add Exp Exp
  | Eq Exp Exp
  | IfElse Exp Exp Exp

-- Valid expressions
ex1 = Add (I 3) (I 4)
ex2 = Add ex1 (I 5)
ex3 = Eq (I 2) (I 7)
ex4 = IfElse ex3 ex1 (I 5)
-- Invalid expressions
ex5 = IfElse (I 1) ex1 ex2
ex6 = IfElse ex3 ex1 ex3

data Result = RI Int | RB Bool deriving (Show)

eval :: Exp -> Result
eval (I i) = RI i
eval (Add a b) = case (eval a, eval b) of
  (RI a', RI b') -> RI $ a' + b'
  (_, _)         -> error "1. Cannot add non-integers"
eval (Eq a b) = case (eval a, eval b) of
  (RI a', RI b') -> RB $ a' == b'
  (_, _)         -> error "2. Forbid equating expressions involving booleans"
eval (IfElse disc a b) = case (eval disc, eval a, eval b) of
  (RB d, RI a', RI b') -> if d then RI a' else RI b'
  (RB d, RB a', RB b') -> if d then RB a' else RB b'
  (RB _, _, _)         -> error "3. Returned expressions must have the same type"
  (_, _, _)            -> error "4. Discriminator must be a boolean"
```

```haskell
view :: Exp -> String
view (I i) = show i
view (Add a b) = "(" ++ view a ++ " + " ++ view b ++ ")"
view (Eq a b) = "(" ++ view a ++ " == " ++ view b ++ ")"
view (IfElse disc a b) = "if " ++ view disc ++ " then " ++ view a ++ " else " ++ view b

data ExpIT a where
  I' :: Int -> ExpIT Int
  Add' :: ExpIT Int -> ExpIT Int -> ExpIT Int
  Eq' :: ExpIT Int -> ExpIT Int -> ExpIT Bool
  IfElse' :: ExpIT Bool -> ExpIT b -> ExpIT b -> ExpIT b
-- Extension with multiplication requires recompilation of all interpreters
  Mult :: ExpIT Int -> ExpIT Int -> ExpIT Int

-- Valid expressions
ex7 = Add' (I' 4) (I' 5)
ex8 = Eq' (I' 4) (I' 5)
ex9 = IfElse' ex8 ex7 (I' 6)
-- Invalid expressions no longer compile, a win!
-- ex10 = Add' (B' True) (I' 4)
-- ex11 = IfElse' ex8 (B' True) (I' 2)

viewIT :: ExpIT a -> String
viewIT (I' i)             = show i
viewIT (Add' a b)         = "(" ++ viewIT a ++ " + " ++ viewIT b ++ ")"
viewIT (Eq' a b)          = "(" ++ viewIT a ++ " == " ++ viewIT b ++ ")"
viewIT (IfElse' disc a b) = "if " ++ viewIT disc ++ " then " ++ viewIT a ++ " else " ++ viewIT b

evalIT :: (Eq a) => ExpIT a -> a
evalIT (I' i)             = i
evalIT (Add' a b)         = evalIT a + evalIT b
evalIT (Eq' a b)          = evalIT a == evalIT b
evalIT (IfElse' disc a b) = if evalIT disc then evalIT a else evalIT b

class AddFT a where
  int :: Int -> a
  add :: a -> a -> a

class EqFT a where
  eq :: Int -> Int -> a
  ifElse :: Bool -> a -> a -> a

instance AddFT Int where
  int = id
  add a b = a + b

instance AddFT String where
  int = show
  add a b = "(" ++ a ++ " + " ++ b ++ ")"

instance EqFT Bool where
  eq = (==)
  ifElse disc a b = if disc then a else b

instance EqFT String where
  eq a b = "(" ++ show a ++ " == " ++ show b ++ ")"
  ifElse disc a b = "if " ++ show disc ++ " then " ++ a ++ " else " ++ b

-- ex12 :: AddFT a => a
ex12 = add (int 3) (int 5)
ex13 = add ex12 (int 2)
ex14 = eq ex12 ex13
ex15 = ifElse ex14 ex12 ex13

viewFT :: String -> String
viewFT = id

evalFT :: Int -> Int
evalFT = id

class MultFT a where
  mult :: a -> a -> a

instance MultFT Int where
  mult = (*)

instance MultFT String where
  mult a b = "(" ++ a ++ " * " ++ b ++ ")"

ex16 = add (mult (int 4) (int 2)) (int 1)
ex17 = mult (int 3) (int 3)
ex18 :: (EqFT a, AddFT a) => a
ex18 = ifElse (eq ex16 ex17) (int 50) (int 51)
```

C.4 Free Monad Iteration

The free monad implementations of Appendices A.9 and A.10 call `iterM` in order to interpret the free monadic program into the `App` interpreter. We outline the steps involved in this process:

1. The compiler inductively resolves an `Eval App` instance from the instances for each member of the coproduct union; the type class constraints derived by `App` make this possible. Conceptually, we now have a function `Free App () -> App ()`.
2. This is function is passed to `iterM` along with the monomorphised program `trackSpaceStation :: Free App ()`.
3. Iteration occurs inside `iterM`, peeling off one instruction at a time and passing it to the provided function. Each iteration produces an `App a` for some `a` decided by the instruction.
4. This is sequenced with the *rest of the program*; passed to `iterM` using monadic combinators such as `>>=` and `>>` defined in the type class instances.
5. This is repeated recursively for each instruction *in* the rest of the program.
6. Eventually, we hit the final instruction: the `Pure` branch of the free monad. We sequence one final `App ()` value before interpreting the result into `IO ()` with `runApp`.

C.5 Composing State and Errors

We examine the compositional semantics of state and errors for every effect system we have covered. Full analysis is in §7.4.2. We take inspiration from a so-called *semantic zoo* performed by King [48]. The examples are categorised into raw `IO`, MTL-style and extensible effects. The full code running every composition in `main`:

```
module StateErrors (main) where

import           Control.Exception     (Exception)
import           Control.Monad.Catch   (MonadCatch, MonadThrow (throwM), catch)
import           Control.Monad.Except  (MonadError, catchError, runExceptT,
                                        throwError)
import           Control.Monad.Ref     (MonadRef (Ref, readRef, writeRef))
import           Control.Monad.State   (MonadState (get, put), evalStateT)
import           Data.IORef            (newIORef)
import           Polysemy              (Members, Sem, runM)
import           Polysemy.AtomicState  (AtomicState, atomicGet, atomicPut,
                                        evalAtomicStateViaState,
                                        runAtomicStateIORef)
import           Polysemy.Error        (Error, runError, throw)
import qualified Polysemy.Error       as P (catch)
import           Prelude               hiding (log)

data MyException = MyException deriving Show
instance Exception MyException

{-- Key:

MTL - Monad Transformer Library
EE  - Extensible Effects
PS  - Pure semantics
CS  - Concurrent semantics
--}
main :: IO ()
main = do
  ref           <- newIORef False
  ioResult      <- putThrowIO ref
  mtlErrorInner <- evalStateT (runExceptT putThrowMTL) False
  mtlStateInner <- runExceptT (evalStateT putThrowMTL False)
  writeRef ref False
  eePSErrorInner <- runM $ evalAtomicStateViaState False $ runError @MyException putThrowEE
  eeCSErrorInner <- runM $ runAtomicStateIORef ref $ runError @MyException putThrowEE
  eePSStateInner <- runM $ runError @MyException $ evalAtomicStateViaState False putThrowEE
  eeCSStateInner <- runM $ runError @MyException $ runAtomicStateIORef ref putThrowEE
  printResult "1. Raw IO" ioResult
  printResult "2. MTL, ExceptT innermost" mtlErrorInner
  printResult "3. MTL, StateT innermost" mtlStateInner
  printResult "4. EE, pure semantics, error innermost" eePSErrorInner
  printResult "5. EE, concurrent semantics, error innermost" eeCSErrorInner
  printResult "6. EE, pure semantics, state innermost" eePSStateInner
```

```
  printResult "7. EE, concurrent semantics, state innermost" eeCSStateInner

printResult :: Show a => String -> a -> IO ()
printResult label result =
  let maxWidth    = 44
      labelLength = length label
      msg         = label ++ replicate (maxWidth - labelLength) ' ' ++ ": " ++ show result
   in print msg

putThrowIO :: (MonadRef m, MonadCatch m) => Ref m Bool -> m Bool
putThrowIO ref =
  writeRef ref True >> throwM MyException `catch` (\(_ :: MyException) -> readRef ref)

putThrowMTL :: (MonadState Bool m, MonadError MyException m) => m Bool
putThrowMTL =
  (put True *> throwError MyException) `catchError` (\(_ :: MyException) -> get)

putThrowEE :: Members '[AtomicState Bool, Error MyException] r => Sem r Bool
putThrowEE =
  (atomicPut True *> throw MyException) `P.catch` (\(_ :: MyException) -> atomicGet)
```

The three functions under test, `putThrowIO`, `putThrowMTL` and `putThrowEE` correspond to our three effect systems. In all three functions, we store `True`, throw an error, immediately catch it and return the state. The initial state provided is always `False`, so that we can see whether the state persisted or not. If the result is `True`, we have state-preserving semantics; otherwise, transactional semantics. The results are as follows:

```
"1. Raw IO                                  : True"
"2. MTL, ExceptT innermost                  : Right True"
"3. MTL, StateT innermost                   : Right False"
"4. EE, pure semantics, error innermost     : Right True"
"5. EE, concurrent semantics, error innermost: Right True"
"6. EE, pure semantics, state innermost     : Right False"
"7. EE, concurrent semantics, state innermost: Right True"
```

For quick reference, the results are aggregated in Figure 7.4.

D Definitions

D.1 Monads and Functors

A type M is a monad if and only if we can define `>>=` and `return` such that laws (i)-(iii) hold:

```
return a >>= f  = f a                     -- (i)   left identity
m >>= return    = m                       -- (ii)  right identity
(m >>= f) >>= g = m >>= (\x -> f x >>= g) -- (iii) associativity
```

Monads can also be characterised by the functions below which constitute a second minimal definition. Noting that `>>=` is equivalent to the composition `join . fmap`, the laws above must again hold.

```
fmap   :: (a -> b) -> M a -> M b
join   :: M (M a) -> M a
return :: a -> M a
```

`fmap` is the defining function for *functors*, referred to frequently throughout Chapter 7. There are only two laws:

```
fmap id      = id              -- (i)  identity
fmap (f . g) = fmap f . fmap g -- (ii) function composition
```

D.2 MaybeT Monad Instance

Assuming the premise that `m` is a monad, we can define a monad instance for the **`MaybeT`** from §5.1 in terms of only the minimal monadic operators:

```
instance Monad m => Monad (MaybeT m) where
  return            = MaybeT . return . Just
  (MaybeT ma) >>= k = MaybeT
    (ma >>= \case
      Just x  -> runMaybeT $ k x
      Nothing -> return Nothing)
```

The pattern used here typifies most instance implementations for transformers: we unwrap the transformer with `runMaybeT`, operate on the nested monads within, and rewrap it at the end using the **`MaybeT`** data constructor. This is a lawful instance, though the proof is omitted.

D.3 Steele's Pseudomonads

Steele's characterisation of monads and pseudomonads are defined in modern Haskell [63]. Though, as he noted, existential types were not available at the time, we can now express them with the `forall` keyword:

```
type UnitFn p q = p -> q
type BindFn p q = q -> (p -> q) -> q
type PseudobindFn p q = forall r. SteeleMonad q r -> q -> (p -> r) -> r

data SteeleMonad p q = SteeleMonad (UnitFn p q) (BindFn p q)
data Pseudomonad p q = Pseudomonad (UnitFn p q) (PseudobindFn p q)

pseudobind :: Pseudomonad p q -> PseudobindFn p q
pseudobind (Pseudomonad _ pbind) = pbind

pseudounit :: Pseudomonad p q -> Unitfn p q
pseudounit (Pseudomonad punit _) = u
```

We define type aliases for the usual monadic operators. A monad gives rise to a **`Pseudomonad`** if the *unit* and *pseudobind* operators can be defined satisfying the same laws of identity and associativity. The composition operator `&` is defined as follows:

```
(&) :: SteeleMonad q r -> Pseudomonad p q -> SteeleMonad p r
(SteeleMonad u b) & (Pseudomonad pu pb) =
  SteeleMonad (u . pu) (\r k -> b r (\p -> pb p k))
```

D.4 Composed Monad Laws

Assume we have two monads M and N. These can be composed if and only if the `join` function described in §4.2.2 can implemented in manner which satisfies the following laws $J(1)$ and $J(2)$. Subscripts indicate which type the operation belongs to, with no subscripts denoting the composed monad. *map* is a common alias for *fmap*, the characteristic functor operation.

$$join_M \,.\, map_M\ join = join \,.\, join_M \qquad J(1)$$

$$join \,.\, map\ (map_M\ join_N) = map_N\ join_N \,.\, join \qquad J(2)$$

D.5 Monad Commutativity

Informally, we can capture the condition for monad commutativity in Haskell code:

```
inOrder :: Monad m => m a -> m b -> (a -> b -> m c) -> m c
inOrder a b f = a >>= \a' -> b >>= \b' -> f a' b'

flipped :: Monad m => m a -> m b -> (a -> b -> m c) -> m c
flipped a b f = b >>= \b' -> a >>= \a' -> f a' b'
```

A monad *m* is commutative if `inOrder` and `flipped` can be substituted for each other in any expression without changing program behaviour.

D.6 Continuation Transformer: ContT

The continuation monad transformer `ContT` is defined in *transformers* [73] as a wrapper round a function containing a monadic computation (the current continuation) as its input.

```
newtype ContT r m a = ContT { runContT :: (a -> m r) -> m r }
```

E Application Logs

All the ISS examples run below are configured with a delay of 5 seconds and a request count of 3. Hence, they produce a list of 2 speeds in chronological order.

E.1 Haskell Examples Run Together

The output from running the main of Appendix A.1.

```
"--------------- 1: Monolithic monad ----------------------------------------------------"
Location (Lat (-42.6159)) (Lon 143.1559) (Timestamp 1666935485)
Location (Lat (-42.8058)) (Lon 143.5408) (Timestamp 1666935490)
"Current speed is: 27274.1km/h"
Location (Lat (-42.9772)) (Lon 143.8929) (Timestamp 1666935495)
"Current speed is: 24796.62km/h"
"No more requests to send to ISS"
"Successfully tracked ISS speed, result is: [27274.1km/h,24796.62km/h]"
"--------------- 2: Transformers, both concrete and abstract -----------------------------"
[Info] Location (Lat (-42.9772)) (Lon 143.8929) (Timestamp 1666935495)
[Info] Location (Lat (-43.1642)) (Lon 144.2825) (Timestamp 1666935501)
[Info] "Current speed is: 22720.74km/h"
[Info] Location (Lat (-43.333)) (Lon 144.6389) (Timestamp 1666935506)
[Info] "Current speed is: 24791.58km/h"
[Info] "No more requests to send to ISS"
[Info] "Successfully tracked ISS speed, result is: [22720.74km/h,24791.58km/h]"
[Info] Location (Lat (-43.333)) (Lon 144.6389) (Timestamp 1666935506)
[Info] Location (Lat (-43.5006)) (Lon 144.9975) (Timestamp 1666935511)
[Info] "Current speed is: 24798.55km/h"
[Info] Location (Lat (-43.6835)) (Lon 145.3944) (Timestamp 1666935516)
[Info] "Current speed is: 27278.57km/h"
[Info] "No more requests to send to ISS"
[Info] "Successfully tracked ISS speed, result is: [24798.55km/h,27278.57km/h]"
"--------------- 3: Transformers improved (ReaderT, Has, tagless final) -----------------"
Location (Lat (-43.6835)) (Lon 145.3944) (Timestamp 1666935516)
Location (Lat (-43.8483)) (Lon 145.7573) (Timestamp 1666935521)
"Current speed is: 24786.24km/h"
Location (Lat (-44.0282)) (Lon 146.1591) (Timestamp 1666935527)
"Current speed is: 22731.15km/h"
"No more requests to send to ISS"
"Successfully tracked ISS speed, result is: [24786.24km/h,22731.15km/h]"
"--------------- 4: Free monads ---------------------------------------------------------"
Location (Lat (-44.0282)) (Lon 146.1591) (Timestamp 1666935527)
Location (Lat (-44.1904)) (Lon 146.5265) (Timestamp 1666935532)
"Current speed is: 24793.39km/h"
Location (Lat (-44.3512)) (Lon 146.8961) (Timestamp 1666935537)
"Current speed is: 24793.23km/h"
"No more requests to send to ISS"
"Successfully tracked ISS speed, result is: [24793.39km/h,24793.23km/h]"
"--------------- 5: Free monads alternate (granular environment + StateT) ---------------"
Location (Lat (-44.3672)) (Lon 146.9332) (Timestamp 1666935537)
Location (Lat (-44.5266)) (Lon 147.3052) (Timestamp 1666935542)
"Current speed is: 24798.29km/h"
Location (Lat (-44.6845)) (Lon 147.6792) (Timestamp 1666935547)
"Current speed is: 24784.99km/h"
"No more requests to send to ISS"
"Successfully tracked ISS speed, result is: [24798.29km/h,24784.99km/h]"
"--------------- 6: Extensible effects, polysemy ----------------------------------------"
Location (Lat (-44.7003)) (Lon 147.7168) (Timestamp 1666935548)
Location (Lat (-44.8568)) (Lon 148.0932) (Timestamp 1666935553)
"Current speed is: 24790.8km/h"
Location (Lat (-45.0118)) (Lon 148.4718) (Timestamp 1666935558)
"Current speed is: 24788.22km/h"
"No more requests to send to ISS"
"Successfully tracked ISS speed, result is: [24790.8km/h,24788.22km/h]"
"--------------- 7: Extensible effects, polysemy alternate ------------------------------"
Location (Lat (-45.0273)) (Lon 148.5098) (Timestamp 1666935558)
Location (Lat (-45.1809)) (Lon 148.8909) (Timestamp 1666935563)
"Current speed is: 24799.77km/h"
Location (Lat (-45.333)) (Lon 149.2741) (Timestamp 1666935568)
"Current speed is: 24793.22km/h"
"No more requests to send to ISS"
"Successfully tracked ISS speed, result is: [24799.77km/h,24793.22km/h]"
```

E.2 Network Failure Error Logs

Removing the network connection produces an error within `IO`. This does not get handled by our program for examples §3.3 and §5.2.

```
*** Exception: HttpExceptionRequest Request {
  host                 = "api.open-notify.org"
  port                 = 80
  secure               = False
  requestHeaders       = [("Accept","application/json")]
  path                 = "/iss-now.json"
  queryString          = ""
  method               = "GET"
  proxy                = Nothing
  rawBody              = False
  redirectCount        = 10
  responseTimeout      = ResponseTimeoutDefault
  requestVersion       = HTTP/1.1
}
 (ConnectionFailure Network.Socket.getAddrInfo (called with preferred socket type/protocol:
 AddrInfo {addrFlags = [AI_ADDRCONFIG], addrFamily = AF_UNSPEC, addrSocketType = Stream,
 addrProtocol = 0, addrAddress = <assumed to be undefined>,
 addrCanonName = <assumed to be undefined>}, host name: Just "api.open-notify.org",
 service name: Just "80"): does not exist (nodename nor servname provided, or not known))
```

E.3 Parsing Failure Error Logs

Breaking the configured URL (for example changing ".json" to ".jso") causes a parsing error since the response is a 404 with an empty body. This error *does* get handled by our program in examples §3.3 and §5.2, evidenced by the log line below.

```
[Info] "Failed to call ISS with error: JSONParseException Request {\n  host ...
```

Formatted for readability, the full exception is:

```
*** Exception: JSONParseException Request {
  host                 = "api.open-notify.org"
  port                 = 80
  secure               = False
  requestHeaders       = [("Accept","application/json")]
  path                 = "/iss-now.jso"
  queryString          = ""
  method               = "GET"
  proxy                = Nothing
  rawBody              = False
  redirectCount        = 10
  responseTimeout      = ResponseTimeoutDefault
  requestVersion       = HTTP/1.1
}
 (Response {responseStatus = Status {statusCode = 404, statusMessage = "Not Found"},
 responseVersion = HTTP/1.1, responseHeaders = [("Server","nginx/1.10.3"),
 ("Date","Mon, 26 Sep 2022 20:13:49 GMT"),("Content-Type","application/json; charset=UTF-8"),
 ("Content-Length","0"),("Connection","keep-alive")], responseBody = (),
 responseCookieJar = CJ {expose = []}, responseClose' = ResponseClose})
 (ParseError {errorContexts = [], errorMessage = "not enough input", errorPosition = 1:1 (0)})
```

E.4 Scala Example

```
Location(33.3502,-116.501,1662048469)
Location(33.1113,-116.208,1662048474)
Current speed is: 27402.094203288456
Location(32.8933,-115.9431,1662048479)
Current speed is: 24919.754641989595
No more requests to send to ISS
Successfully tracked ISS speed, result is: List(27402.094203288456, 24919.754641989595)
```

F International Space Station API JSON Payload


```
{
  "message": "success",
  "timestamp": 1659130098,
  "iss_position": {
    "longitude": "18.1161",
    "latitude": "20.6183"
  }
}
```

Bibliography

[1] P. Wadler, "The essence of functional programming," POPL '92, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 1992.

[2] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 333–343, 1995.

[3] M. P. Jones, "Functional programming with overloading and higher-order polymorphism," in *International School on Advanced Functional Programming*, pp. 97–136, Springer, 1995.

[4] R. K. Merton, "The Matthew effect in science, ii: Cumulative advantage and the symbolism of intellectual property," *isis*, vol. 79, no. 4, pp. 606–623, 1988.

[5] B. Nystrom, "What Color is Your Function?," 2015. `https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function` [accessed 30/09/2022].

[6] G. Plotkin and M. Pretnar, "Handlers of algebraic effects," in *European Symposium on Programming*, pp. 80–94, Springer, 2009.

[7] "Rust Tokio Asychronous Runtime Documentation." `https://docs.rs/tokio/latest/tokio/task/` [accessed 30/09/2022].

[8] A. Sabry, "What is a purely functional language?," *Journal of Functional Programming*, vol. 8, no. 1, pp. 1–22, 1998.

[9] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pp. 61–78, 1990.

[10] S. L. Peyton Jones and P. Wadler, "Imperative functional programming," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 71–84, 1993.

[11] Typelevel, "Cats Effect: Thread Model," 2022. `https://typelevel.org/cats-effect/docs/thread-model` [accessed 30/09/2022].

[12] G. Plotkin and M. Pretnar, "A logic for algebraic effects," in *2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pp. 118–129, IEEE, 2008.

[13] "Eff Research Language." `https://www.eff-lang.org/` [accessed 16/10/2022].

[14] "Frank Research Language." `https://github.com/frank-lang/frank` [accessed 30/09/2022].

[15] "Koka Research Language." `https://koka-lang.github.io/koka/doc/index.html` [accessed 30/09/2022].

[16] "Effekt Research Language." `https://effekt-lang.org/` [accessed 16/10/2022].

[17] "Idris Research Language." Closer to being production-ready, but still advertised as a research language. `https://docs.idris-lang.org/en/latest/index.html` [accessed 30/09/2022].

[18] "Unison language." Available at: `https://www.unison-lang.org/learn/`. Accessed at 25/10/2022.

[19] "Haskell EvEff Library." `https://github.com/xnning/EvEff` [accessed 30/09/2022].

[20] "Scala Effekt Library." `https://b-studios.de/scala-effekt/` [accessed 30/09/2022].

[21] "Multicore OCaml Extension." `https://github.com/ocaml-multicore/ocaml-multicore/wiki` [accessed 30/09/2022].

[22] O. Kiselyov, A. Sabry, and C. Swords, "Extensible effects: an alternative to monad transformers," *ACM SIGPLAN Notices*, vol. 48, no. 12, pp. 59–70, 2013.

[23] O. Kiselyov and H. Ishii, "Freer monads, more extensible effects," *ACM SIGPLAN Notices*, vol. 50, no. 12, pp. 94–105, 2015.

[24] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys (CSUR)*, vol. 21, no. 3, pp. 359–411, 1989.

[25] J. Backus, “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs,” *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[26] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, “A history of Haskell: being lazy with class,” in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12–1, 2007.

[27] D. K. Gifford and J. M. Lucassen, “Integrating functional and imperative programming,” in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 28–38, 1986.

[28] P. Roe, “An imperative language with read/write type modes,” in *Annual Asian Computing Science Conference*, pp. 254–267, Springer, 1997.

[29] G. M. Bierman and M. J. Parkinson, “Effects and effect inference for a core Java calculus,” *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 8, pp. 82–107, 2003.

[30] D. Syme, T. Petricek, and D. Lomov, “The F# asynchronous programming model,” in *International Symposium on Practical Aspects of Declarative Languages*, pp. 175–189, Springer, 2011.

[31] J.-P. Talpin and P. Jouvelot, “The type and effect discipline,” *Information and computation*, vol. 111, no. 2, pp. 245–296, 1994.

[32] D. Marino and T. Millstein, “A generic type-and-effect system,” in *Proceedings of the 4th international workshop on Types in language design and implementation*, pp. 39–50, 2009.

[33] P. Hudak and R. S. March, “On the expressiveness of purely-functional I/O systems,” tech. rep., Citeseer, 1989.

[34] L. McLoughlin and E. Hayes, “Imperative effects from a pure functional language,” in *Functional Programming*, pp. 157–169, Springer, 1990.

[35] D. A. Turner, “Miranda: A non-strict functional language with polymorphic types,” in *Conference on Functional Programming Languages and Computer Architecture*, pp. 1–16, Springer, 1985.

[36] D. A. Turner, “Some history of functional programming languages,” in *International Symposium on Trends in Functional Programming*, pp. 1–20, Springer, 2012.

[37] J. T. O’Donnell, “Dialogues: A basis for constructing programming environments,” *ACM SIGPLAN Notices*, vol. 20, no. 7, pp. 19–27, 1985.

[38] S. Thompson, “Interactive functional programs: a method and a formal semantics,” 1987. Technical Report No. 48, UKC Computing Laboratory.

[39] H. Abelson and G. J. Sussman, *Structure and interpretation of computer programs*. The MIT Press, 1996.

[40] A. Filinski, “Representing monads,” in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 446–457, 1994.

[41] A. Filinski, “Representing layered monads,” POPL ’99, p. 175–188, Association for Computing Machinery, 1999.

[42] E. Moggi, “Computational lambda-calculus and monads,” in *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23, IEEE Comput. Soc. Press, 1989.

[43] E. Moggi, *An abstract view of programming languages*. University of Edinburgh. Department of Computer Science. Laboratory for Foundations of Computer Science, 1990.

[44] D. Scott, *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[45] D. S. Scott and C. Strachey, *Toward a mathematical semantics for computer languages*, vol. 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

[46] E. Moggi, “Notions of computation and monads,” *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[47] S. P. Jones, “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell,” in *Engineering theories of software construction*, pp. 47–96, Press, 2001.

[48] A. King, "The Effect Semantics Zoo." https://github.com/lexi-lambda/eff/blob/master/notes/semantics-zoo.md [accessed 24/10/2022].

[49] A. D. Gordon, *Functional programming and input/output*. No. 8, Cambridge University Press, 1994.

[50] S. Lindley, C. McBride, and C. McLaughlin, "Do be do be do," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 500–514, ACM, 2017.

[51] Typelevel, "Scala Library: *Cats Effect: The pure asynchronous runtime for Scala*," 2022. https://typelevel.org/cats-effect/ [accessed 30/09/2022].

[52] ZIO, "Scala Library: *ZIO: Type-safe, composable asynchronous and concurrent programming for Scala*," 2022. https://zio.dev/ [accessed 14/10/2022].

[53] N. Bergey, "Open Notify API Documentation: International Space Station Location," 2022. http://open-notify.org/Open-Notify-API/ISS-Location-Now/ [accessed 30/09/2022].

[54] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, p. 196–es, 1996.

[55] D. Ghosh, *DSLs in action*. Simon and Schuster, 2010.

[56] O. Kiselyov, "Effects without monads: Non-determinism – back to the meta language," *Electronic Proceedings in Theoretical Computer Science*, vol. 294, pp. 15–40, 2019.

[57] J. Backus, J. H. Williams, and E. L. Wimmers, *An Introduction to the Programming Language FL*, p. 219–247. USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[58] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. O'Reilly Media, Inc., 1st ed., 2008.

[59] J. C. Reynolds, "User-defined types and procedural data structures as complementary approaches to data abstraction," in *Programming Methodology*, pp. 309–317, Springer, 1978.

[60] P. Wadler, "The Expression Problem." https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt [accessed 30/09/2022].

[61] N. Wu and T. Schrijvers, "Fusion for free," in *International Conference on Mathematics of Program Construction*, pp. 302–322, Springer, 2015.

[62] A. King, "GHC Proposal: Adding a Primitive for Delimited Continuations," 2021. https://github.com/ghc-proposals/ghc-proposals/pull/313 [accessed 30/09/2022].

[63] G. L. Steele Jr, "Building interpreters by composing monads," in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 472–492, 1994.

[64] J. Beck, "Distributive laws," in *Seminar on triples and categorical homology theory*, pp. 119–140, Springer, 1969.

[65] M. Jones and L. Duponcheel, "Composing monads." Research Report YALEU/DCS/RR-1004, Yale University Department of Computer Science.

[66] nLab, "Pointed Endofunctor," 2021. https://ncatlab.org/nlab/show/pointed+endofunctor [accessed 30/09/2022].

[67] M. P. Jones, "A system of constructor classes: Overloading and implicit higher-order polymorphism," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, p. 52–61, Association for Computing Machinery, 1993.

[68] D. Espinosa, "Building interpreters by transforming stratified monads," *Unpublished manuscript, ftp from altdorf.ai.mit. edu: pub/dae*, 1994. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.6588&rep=rep1&type=pdf [accessed 30/09/2022].

[69] D. J. King and P. Wadler, "Combining monads," in *Functional Programming, Glasgow 1992*, pp. 134–143, Springer London, 1993.

[70] W. Swierstra, "Data types à la carte," *Journal of Functional Programming*, vol. 18, no. 4, p. 423–436, 2008.

[71] O. Kammar, S. Lindley, and N. Oury, "Handlers in action," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, p. 145–158, Association for Computing Machinery, 2013.

[72] E. Brady, "Programming and reasoning with algebraic effects and dependent types," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pp. 133–144, 2013.

[73] "Haskell Library. *transformers*: Concrete Functor and Monad Transformers." `https://hackage.haskell.org/package/transformers` [accessed 30/09/2022].

[74] "Haskell Library. *mtl*: Monad Classes, Using Functional Dependencies." `https://hackage.haskell.org/package/mtl` [accessed 30/09/2022].

[75] "Haskell Library. *rio*: A standard library for Haskell." `https://hackage.haskell.org/package/rio` [accessed 14/10/2022].

[76] M. Snoyman, "The ReaderT Design Pattern," 2016. `https://www.fpcomplete.com/blog/2017/06/readert-design-pattern` [accessed 30/09/2022].

[77] "Haskell Wiki: Error vs. Exception," 2019. `https://wiki.haskell.org/Error_vs._Exception` [accessed 14/10/2022].

[78] M. Sulzmann, "Overlapping Instances and Coherence," 2006. `https://mail.haskell.org/pipermail/haskell-cafe/2006-April/015367.html` [accessed 30/09/2022].

[79] "Scala 3 Type Class Derivation." `https://docs.scala-lang.org/scala3/reference/contextual/derivation.html` [accessed 30/09/2022].

[80] F. Slama, *"Automatic generation of proof terms in dependently typed programming languages"*. PhD thesis, University of St Andrews, UK, 2018. Available: `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.759365`.

[81] J. Gibbons and R. Hinze, "Just do it: Simple monadic equational reasoning," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, p. 2–14, Association for Computing Machinery, 2011.

[82] "Haskell Wiki: ListT," 2021. `https://wiki.haskell.org/ListT_done_right` [accessed 30/09/2022].

[83] "Haskell Wiki: Amb," 2008. `https://wiki.haskell.org/Amb` [accessed 30/09/2022].

[84] "Haskell Wiki: ContStuff and the ChoiceT Transformer," 2021. `https://wiki.haskell.org/Contstuff#Choice.2Fnondeterminism` [accessed 30/09/2022].

[85] A. King, "Effects for Less," 2020. [Online Video]. `https://youtu.be/0jI-AlWEwYI` [accessed 30/09/2022].

[86] "Contextual Abstractions in Scala 3." `https://dotty.epfl.ch/docs/reference/contextual/index.html` [accessed 30/09/2022].

[87] A. King, "Lifts for Free: Making MTL Typeclasses Derivable." `https://lexi-lambda.github.io/blog/2017/04/28/lifts-for-free-making-mtl-typeclasses-derivable/` [accessed 30/09/2022].

[88] G. Hutton, "Fold and unfold for program semantics," in *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, 1998.

[89] O. Kiselyov, *Typed Tagless Final Interpreters*, pp. 130–174. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[90] A. King, "Parse, Don't Validate," 2019. `https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/` [accessed 03/10/2022].

[91] J. Carette, O. Kiselyov, and C.-c. Shan, "Finally tagless, partially evaluated," in *Programming Languages and Systems*, pp. 222–238, Springer Berlin Heidelberg, 2007.

[92] L. Jacobowitz, "Optimizing Tagless Final - Saying Farewell to Free," 2017. https://typelevel.org/blog/2017/12/27/optimizing-final-tagless.html [accessed 03/10/2022].

[93] M. Snoyman, "*rio*: A standard library." https://www.fpcomplete.com/haskell/library/rio/ [accessed 11/10/2022].

[94] "Haskell Library. *transformers*: Dependency injection for records-of-functions." https://hackage.haskell.org/package/dep-t [accessed 07/10/2022].

[95] "Haskell Library. *has-transformers*: This library 'Has' transformers." https://hackage.haskell.org/package/has-transformers [accessed 12/10/2022].

[96] J. De Goes, "Beautiful, Simple, Testable Functional Effects for Scala." https://degoes.net/articles/zio-environment [accessed 11/10/2022].

[97] "Context Functions in Scala 3." https://dotty.epfl.ch/docs/reference/contextual/context-functions.html [accessed 27/10/2022].

[98] "Haskell Library. *stm*: Software Transactional Memory." https://hackage.haskell.org/package/stm [accessed 08/10/2022].

[99] Tim Spence, "Scala Library: *Cats STM*," 2022. https://timwspence.github.io/cats-stm/ [accessed 08/10/2022].

[100] S. Peyton Jones, *Beautiful concurrency*. O'Reilly, beautiful code ed., January 2007.

[101] "Haskell Library. *monad-control*: Lift control operations, like exception catching, through monad transformers." https://hackage.haskell.org/package/monad-control [accessed 13/10/2022].

[102] A. King, "Demystifying monadbasecontrol." https://lexi-lambda.github.io/blog/2019/09/07/demystifying-monadbasecontrol/ [accessed 10/10/2022].

[103] S. O'Brien, "Haskell Library. *layers*: Modular type class machinery for monad transformer stacks.." https://hackage.haskell.org/package/layers/docs/Documentation-Layers-Overview.html [accessed 13/10/2022].

[104] "Haskell Library. *data-has*: Simple extensible product." https://hackage.haskell.org/package/data-has [accessed 07/10/2022].

[105] "Haskell Library. *lens*: Lenses, Folds and Traversals." https://hackage.haskell.org/package/lens [accessed 07/10/2022].

[106] O. Kiselyov, "Extensible Effects vs Data types à la Carte," 2014. https://okmij.org/ftp/Haskell/extensible/extensible-a-la-carte.html [accessed 30/09/2022].

[107] R. Bjarnason, "Constraints Liberate, Liberties Constrain," 2016. [Online Video]. https://www.youtube.com/watch?v=GqmsQeSzMdw [accessed 14/10/2022].

[108] "Haskell Library. *polysemy*: Higher-order, low-boilerplate free monads.." https://hackage.haskell.org/package/polysemy [accessed 24/10/2022].

[109] "Haskell Library. *fused-effects*: A fast, flexible, fused effect system.." https://hackage.haskell.org/package/fused-effects [accessed 14/10/2022].

[110] C. Lüth and N. Ghani, "Composing monads using coproducts," in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, p. 133–144, Association for Computing Machinery, 2002.

[111] R. Cartwright and M. Felleisen, "Extensible denotational language specifications," in *International Symposium on Theoretical Aspects of Computer Software*, pp. 244–272, Springer, 1994.

[112] A. Bauer and M. Pretnar, "Programming with algebraic effects and handlers," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 108–123, 2015.

[113] "JavaScript Library. *effects.js*: Algebraic effects in javascript with scoped handlers, multishot delimited continuations and do notation." Available at: https://nythrox.github.io/effects.js. Accessed at 25/10/2022.

[114] "Typescript Library. *fx-ts*: Capabilities & Effects for TypeScript." Available at: https://github.com/briancavalier/fx-ts#documentation. Accessed at 25/10/2022.

[115] "Haskell Library. *extensible-effects*: An Alternative to Monad Transformers." https://hackage.haskell.org/package/extensible-effects [accessed 16/10/2022].

[116] "Haskell Library. *freer-simple*: A friendly effect system for Haskell.." https://hackage.haskell.org/package/freer-simple [accessed 25/10/2022].

[117] N. Wu, T. Schrijvers, and R. Hinze, "Effect handlers in scope," in *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2014.

[118] "Haskell Library. *eff*: screaming fast extensible effects for less." https://github.com/lexi-lambda/eff [accessed 24/10/2022].

[119] S. Maguire, "Freer Monads, More Better Programs." Available at: https://reasonablypolymorphic.com/blog/freer-monads. Accessed at 25/10/2022.

[120] "Scala Library. *eff*: Extensible Effects in Scala." Available at: https://atnos-org.github.io/eff/. Accessed at 25/10/2022.

[121] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala," *Journal of Functional Programming*, vol. 30, p. e8, 2020.

[122] A. Granin, "Hierarchical Free Monads: The Most Developed Approach In Haskell," 2020. https://github.com/graninas/hierarchical-free-monads-the-most-developed-approach-in-haskell [accessed 20/10/2022].

[123] S. Maguire, "Polysemy Internals: Freer Interpretations of Higher-Order Effects," 2019. https://reasonablypolymorphic.com/blog/freer-higher-order-effects/ [accessed 24/10/2022].

[124] "Haskell Library. *in-other-words*: A higher-order effect system where the sky's the limit.." https://hackage.haskell.org/package/in-other-words [accessed 24/10/2022].

[125] "Scala Library. *fs2*: Functional, effectful, concurrent streams for Scala.." Available at: https://fs2.io. Accessed at 26/10/2022.

[126] "Scala Library. *doobie*: A pure functional JDBC layer for Scala.." Available at: https://tpolecat.github.io/doobie/. Accessed at 26/10/2022.

[127] S. Maguire, "PPolysemy: Mea Culpa," 2020. https://reasonablypolymorphic.com/blog/mea-culpa/ [accessed 27/10/2022].

[128] D. Spiewak, "Understanding Comparative Benchmarks." Available at: https://gist.github.com/djspiewak/f4cfc08e0827088f17032e0e9099d292. Accessed at 26/10/2022.

[129] "Haskell Language Server. *haskell-language-server*: Official Haskell Language Server implementation.." Available at: https://haskell-language-server.readthedocs.io. Accessed at 26/10/2022.

[130] "Scala Language Server. *metals*: Scala language server with rich IDE features.." Available at: https://scalameta.org/metals/. Accessed at 26/10/2022.

[131] J. Voigtländer, "Asymptotic improvement of computations over free monads," in *Mathematics of Program Construction*, (Berlin, Heidelberg), pp. 388–403, Springer Berlin Heidelberg, 2008.

[132] E. Kmett, "Free Monads for Less (Part 2 of 3): Yoneda." Available at: http://comonad.com/reader/2011/free-monads-for-less. Accessed at 26/10/2022.

[133] "Haskell Library. *free*: Monads for free.." Available at: https://hackage.haskell.org/package/free. Accessed at 26/10/2022.

[134] D. Spiewak, "Quick Tips for Fast Code on the JVM." Available at: https://gist.github.com/djspiewak/464c11307cabc80171c90397d4ec34ef. Accessed at 26/10/2022.

[135] S. Diehl, "Exotic Programming Ideas: Part 3 (Effect Systems)," 2020. https://www.stephendiehl.com/posts/exotic03.html [accessed 27/10/2022].

[136] "Capture Checking: Capabilities in Scala." https://www-dev.scala-lang.org/scala3/reference/experimental/cc.html [accessed 27/10/2022].

[137] "Explaining the Problem with Orphan Instances," 2010. https://stackoverflow.com/a/3079748/7524374 [accessed 30/09/2022].