Informatics

Opt-in Protokolltypen für Effektsysteme in Haskell

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Hannes Siebenhandl, BSc
Matrikelnummer 01327006

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Inf. Dr.rer.nat. Jens Knoop

Wien, 1. Dezember 2023

Hannes Siebenhandl   Jens Knoop

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Opt-in Protocol Types for Effect Systems in Haskell

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Hannes Siebenhandl, BSc
Registration Number 01327006

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Inf. Dr.rer.nat. Jens Knoop

Vienna, 1st December, 2023

Hannes Siebenhandl   Jens Knoop

Erklärung zur Verfassung der Arbeit

Hannes Siebenhandl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Dezember 2023

Hannes Siebenhandl

Acknowledgements

I am deeply thankful to my supervisor Univ.Prof. Jens Knoop for his valuable support and thoughtful advice during this thesis. Special thanks to Jana Chadt and Samuel Pilz for their repeated constructive criticism, insightful discussions and technical and moral support. Finally, I thank my parents for their support throughout my studies.

Kurzfassung

Algebraische Effektsysteme sind eine vielversprechende Alternative zu Monadentransformatoren, indem Effektsysteme die Seiteneffekte von Prozeduren in der Typensignatur feingranular dokumentieren. Obwohl existierende algebraische Effektsysteme es erlauben Seiteneffekte detailliert mitzuverfolgen, gibt es derzeit keine Möglichkeit, komplexe Vor- und Nachbedingungen automatisiert vom Übersetzer verifizieren zu lassen. Dieses Problem addressieren wir in dieser Arbeit und stellen `preff` vor, das erste in Haskell geschriebene algebraische Effektsystem, welches traditionelle algebraische Effekte mit parameterisierten, algebraischen Effektsystemen vereint. Parameterisierte algebraische Effektsysteme erlauben es zur Übersetzungszeit komplexe Vor- und Nachbedingungen im Ausführungskontext des Effekts zu verifizieren. Diese Kombination ermöglicht es, algebraische Effekte zu definieren und zu benutzen, aber auch die Mächtigkeit von parameterisierten Effekten zu nutzen, wenn es vorteilhaft ist. Um die Ausdrucksstärke und Anwendbarkeit von `preff` zu demonstrieren, implementieren wir Sitzungstypen in weniger als 120 Zeilen Quellcode und vergleichen in einer Reihe von Mikroexperimenten die Laufzeit mit fortgeschrittenen Effektsystemen in Haskell. Unsere Evaluation zeigt, dass `preff` ausgezeichnet geeignet ist, um komplexe Programmkontrollflüsse zu implementieren und die Laufzeit in der Praxis kompetitiv zu der aktueller Effektsysteme ist.

Abstract

Algebraic effect systems are a promising alternative to existing monad transformer approaches providing fine-grained tracking of side effects in the type signature of procedures. Even though existing algebraic effect systems allow describing a contract for tracking side effects, there is no way to capture more complex pre- and postconditions that can be verified by the compiler rather than the programmer. To address this shortcoming, we introduce `preff`, the first algebraic effect system library in Haskell that combines traditional algebraic effects and parameterised algebraic effects which enforce pre- and postconditions in the effectful code at compile-time. This allows developers to define and use algebraic effects, while also opting into parameterised effects to great effect. To demonstrate the expressiveness and real-world applicability, we use `preff` to implement session types in less than 120 lines of code and compare the run-time performance to state-of-the-art effect system libraries in a series of microbenchmarks. Our evaluation shows that `preff` is capable of expressing complex control-flow patterns comparable to related session type libraries and has competitive run-time performance showcasing its applicability in practice.

Contents

CHAPTER 1

Introduction

Development and maintenance of software within time and budget constraints is among the greatest challenges in software engineering. However, the need for producing software quickly is at odds with producing maintainable software, and developers often find themselves trying to balance between writing code quickly and writing it in a maintainable way. Maintainable components use separation of concerns and can be tested with less effort as a result, which in turn helps to maintain the quality of written software. Having a good overview of the control flow of a program is important to developers for understanding the causality of computations and, consequentially, maintaining the code quality when extending existing functionality. Therefore, the available control flow primitives can help developers to continuously improve and maintain software. At last, tracking side effects explicitly in the signature of procedures makes it easier for developers, and reviewers, to understand the purpose of components, and how to use them.

Algebraic effect systems promise a way of structuring programs that is easy to use, easy to understand and extensible. In the last decades, algebraic effect systems have been intensely studied and introduced into many programming languages including Haskell. Haskell is a purely functional programming language (Marlow et al., 2010), powered by the de facto standard compiler Glasgow Haskell Compiler (GHC) (Peyton Jones et al., 1993). It allows developers to implement algebraic effect systems without requiring special language support for algebraic effect systems, contrary to languages such as Effekt (Brachthäuser et al., 2020) and Koka (Leijen, 2014) that support user-defined algebraic effects as a native language feature.

In purely functional languages like Haskell, there is no concept of a “procedure” since everything is a function, meaning, given the same input parameters it will produce the same output. However, there are functions that must have side effects, since they affect the outside world or examine external sources. For example, reading the system clock twice with a second delay in between has to yield two different results. In Haskell, it is impossible to have a function of the type `() -> Time`, which produces a different

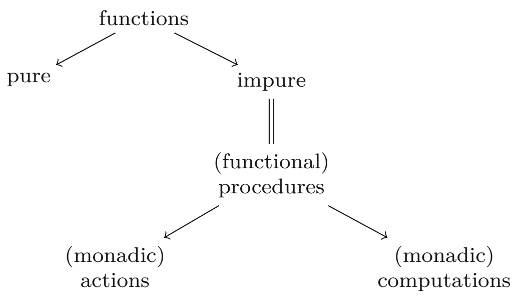

Figure 1.1: Terminology overview of *pure*/*impure* functions.

result on each invocation. How can a Haskell program then have any meaningful impact on the world, if we cannot produce results based on external sources? Essentially, by computing and hiding side effects inside monads, giving the illusion of functions that produce different results even though their input parameters remain unchanged. We give the name *impure* to such functions since they seem to no longer be defined solely by their input parameters, while functions are *pure*, if the output value is fully determined by its input parameters. A monad in Haskell allows writing seemingly procedural code, just like in most mainstream imperative programming languages, using the **do**-notation. The best-known monad is the **IO** monad. In the **IO** monad, we use functions that have side effects, such as writing to a file, reading the system clock or interacting with a database.

We refer to *impure* functions which use monads via **do**-notation as *functional procedures* or simply *procedures*. Functions are *pure* when the type signature indicates that there are no unobservable side effects. Our separation between *pure* and *impure* is conceptual and not sharp: the main purpose is to express expectations about the code. For readability of this thesis, we further split *functional procedures* into *monadic actions*, and *monadic computations*. We refer to code that serves a specific purpose in an application as *monadic actions*. Such code serves often the implementation of business logic, i.e. logic to accommodate for the needs and usage in businesses. It is not expected that business logic is reused in other applications. For example, a functional procedure that requires the existence of a database with a specific table layout is business logic, as such we may refer to it as a monadic action. A monadic computation is meant to be reusable in many different contexts, especially in other applications or libraries. Like the separation of pure and impure functions, the distinction between *monadic actions* and *monadic computations* is conceptual and not sharp. As a rule of thumb, monadic code that can be factored into its own logical unit for reuse is a *(monadic) computation*, and a concrete example or procedure that implements business logic is a *(monadic) action*. In Figure 1.1, we show a schematic hierarchy of the terminology used throughout this thesis.

1.1 Algebraic Effect Systems

Algebraic effect systems enable fine-granular tracking of side effects and control flows, whereas monads permit a binary view between pure and impure computations. Each procedure is annotated with a set of effects called the *effect signature*, or sometimes the *effect list* of a procedure. The effect signature of a procedure is the set of effects that can be executed in said procedure. Further, a procedure 'A' can invoke another effectful procedure 'B' if the effect signature of 'A' is a superset of the effect signature of 'B'. This composes well in practice, and allows each procedure to depend exclusively on the effects it requires for its implementation.

Algebraic effect systems can intuitively be viewed as a client-server architecture, where the client sends a request to the server, whereupon the server processes the request and responds accordingly. However, contrary to usual client-server architectures, in effect systems there is not one definite server, but multiple so-called *algebraic effect handlers*, in short *effect handlers* or simply *handlers*, that respond to one effect only and, composed together, form the server. An *algebraic effect* represents one or more actions that can be performed in a code block and produce results, but contains no details regarding how to perform the action, or how to produce the final result. This is left to the algebraic effect handler which interprets the actions of the respective effect. For each effect, there may be more than one interpretation, i.e. algebraic effect handler.

In Kiselyov et al. (2013), the authors show that effect systems enhance the composability of software and allow for local reasoning. Due to the composability promises, effect systems provide a suitable means to decoupling the implementation from a developer's intent, allowing to specify procedures service-agnostically and declaratively. Additionally, they promote code reuse and simplify testing, since they enable developers to create unit tests for procedures by specifying effect handlers that do not rely on external services. Effect systems are capable of expressing complex control flow patterns, such as *coroutines* or *exceptions*, without requiring built-in programming language support.

A parameterised effect is like an ordinary effect but is enriched with additional type details, a so-called pre- and postcondition. The precondition is a type-level term which describes a prerequisite that needs to hold before the effect can be executed. Accordingly, the postcondition is a type-level term that holds after effect execution. This allows parameterised effects to express complex type-level proofs and enables developers to omit certain run-time validations. Punchihewa and Wu (2021) use parameterised effects to ensure compile-time correctness of parallel sorting algorithms.

Scoped effects, also referred to as higher-order effects, are effects that scope over a monadic region. For example, consider the "error" effect which defines exceptional values that can be thrown by a procedure. When an exception is thrown, normal program execution is interrupted and terminated. A scoped "error" effect can then avoid program termination by catching exceptions that are thrown in a specific code block, also called a monadic region. In other words, a scoped effect *captures* or *scopes over* a monadic region in which the behaviour of the effect can be altered. While scoped effects increase the flexibility of an algebraic effect system library, they also incur difficulties for effect handlers of other effects. Effect handlers need to handle effects within monadic regions that have been scoped over by a scoped effect, but the structure of a scoped effect is opaque to the effect handler. Thus, the effect handler needs to be *weaved* through the scoped regions of the scoped effect. Without it, intermediate results would be lost, hampering the usefulness of scoped effects. The state-of-the-art solution is to define a `weave` function that each scoped effect implements individually to weave execution context through the scoped region it captures. Scoped effects and `weave` are described in detail by Wu et al. (2014) and formalised by Piróg et al. (2018).

To conclude, algebraic effect systems in Haskell define an effect monad that is used to define and run programs. The effect monad is the glue that allows tracking of side effects, execution of effects and implementation of effect handlers. As such, the implementation of the effect monad is of paramount importance, affecting the usability, performance and supported features of the algebraic effect system library. Multiple implementations have been explored in literature, such as the so-called free (Kiselyov et al., 2013) and freer monad (Kiselyov and Ishii, 2015), evidence translation (Xie and Leijen, 2020) and delimited continuations (Kammar et al., 2013).

1.2 Related Work

Effect systems are a vibrant research area, meant to tackle conflicting goals, such as ease of use, expressiveness and high performance. Therefore, there exist many state-of-the-art effect system libraries in Haskell. Many of them have different trade-offs in terms of usability, performance and features. We briefly introduce the effect system libraries this thesis took the most inspiration from, as well as other notable effect system libraries. As Kiselyov et al. (2013) were among the first to introduce a working effect system library in Haskell, most presented effect system libraries are based on their contributions.

`freer-simple`

The algebraic effect system library `freer-simple` is based on the contributions by Kiselyov and Ishii (2015). It is less complex than other state-of-the-art effect system libraries and does not support scoped effects as first-class concepts. The implementation of the effect monad **`Eff`** uses a freer monad encoding. Further, it employs many optimisation techniques to improve the complexity of the monadic sequencing operations (Voigtländer, 2008) and focuses on developer usability.

`fused-effects`

Wu and Schrijvers (2015), Schrijvers et al. (2019), and Wu et al. (2014) provide the foundations of the algebraic effect system library `fused-effects`. It supports scoped effects and provides powerful effect handler strategies to enable developers to define and use effects with minimal overhead. For example, the representation of the effect monad can be fine-tuned according to performance and ease-of-use requirements. Furthermore, it introduces a sophisticated *fusion* framework, that allows multiple effect handlers to be fused into one. Effect handlers essentially need to traverse the full abstract syntax tree (AST) of a program to replace an effect operation with the corresponding monadic computation. Since programs in real-world applications tend to grow, traversing the AST can be costly. Fusing multiple effect handlers reduces the cost of traversing the AST and promises a significant performance speed-up.

`polysemy`

Similar to `fused-effects`, the algebraic effect system library `polysemy` is based on the contributions by Wu et al. (2014). The library `polysemy` focuses on the user experience and ease of use. It differentiates between scoped effects and "simple" effects, with the goal of guiding users to prefer a simpler interface. Additionally, it adds many custom type errors that contextualise type errors to simplify the usage. To increase adoption of algebraic effect system libraries, `polysemy` is the first library to provide the GHC type checker plugin `polysemy-plugin`[1]. It enables the compiler to infer the types of effect operations in more cases, avoiding additional type hints from the developer. Such type checker plugins have become a common practice for algebraic effect system libraries in Haskell. To improve the asymptotic complexity of the monadic sequencing operation, `polysemy` uses a final encoding of the freer monad (Carette et al., 2007).

`safe-mutations`

The algebraic effect system library `safe-mutations`[2] supports parameterised effects and is implemented by Punchihewa and Wu (2021), based on the contributions of Piróg et al. (2018). A parameterised effect enables developers to encode type-level proofs in the effect monad of `safe-mutations`. Whereas user-defined algebraic parameterised effects can be implemented, only a single effect can be used in `safe-mutations`.

`effectful`

The algebraic effect system library `effectful`[3] focuses on real-world applications. It uses the so-called **`ReaderT`** design pattern as its effect monad implementation, to create a high-performance algebraic effect system library. GHC optimises the resulting code more reliably and efficiently. However, this entails that many effects, such as **`State`** or **`Error`**, have to be implemented using side effects. These effects rely on other primitives of GHC such as **`IORef`** and exceptions. Additionally, some effects can be interpreted in only one way, making them *statically* dispatched effects. Whereas this improves the performance for built-in effects, it also renders them less flexible.

[1] `https://hackage.haskell.org/package/polysemy-plugin`
[2] `https://github.com/hashanp/safe-mutations`
[3] `https://hackage.haskell.org/package/effectful`

`cleff`

The algebraic effect system library `cleff`[4] is very similar to `effectful`. It also uses the **`ReaderT`** transformer over the **`IO`** monads and implements many effects in terms of existing features provided in GHC. However, it provides a more flexible interface and does not support statically dispatched effects.

`eff`

The experimental algebraic effect system library `eff`[5] is implemented on top of delimited continuations. It promises a high-performance effect system library that fixes issues of existing libraries, as shown by King (2020). However, it remains an experimental library that requires support for delimited continuations from GHC[6].

`mtl` - Monad Transformer Library

Monad transformers are one of the main approaches for expressing side effects in a modular way. First introduced in functional programming languages by King and Wadler (1993), they were popularised in Haskell by Liang et al. (1995), remaining a foundational tool in Haskell software development until today. The Haskell packages `mtl`[7] and `transformers`[8] provide the most common monad transformer implementations, and many other libraries are built on top of them to provide crucial features. Even the implementation of GHC relies on monad transformers, and provides additional performance optimisations for monad transformer primitives. Due to their maturity and availability, many software developers utilise them to express side effects. According to Kiselyov et al. (2013), algebraic effect systems are more flexible than monad transformers, as they naturally enable adding effects to arbitrary monad structures. Further, they show that some interleaving effects cannot be expressed in `mtl` at all.

The relation between monad transformers and algebraic effect systems has been studied by Schrijvers et al. (2019). They identify a class of modular algebraic effects and effect handlers that can be embedded into a monad transformer and vice versa. It remains unclear whether this generalises to effects with scopes and their categorical formalisation defined by Piróg et al. (2018). Further, Wu and Schrijvers (2015) observe another close connection between effect system libraries based on free monad encodings and monad transformers: monad transformers are fused forms of effect handlers. In other words, monad transformer interpreters fuse naturally and have improved performance characteristics over free monad encodings.

The state-of-the-art research addresses different combinations of tradeoffs of the conflicting goals of algebraic effect systems. However, to the best of our knowledge, no algebraic effect

[4]`https://hackage.haskell.org/package/cleff`
[5]`https://github.com/hasura/eff`
[6]`https://github.com/ghc-proposals/ghc-proposals/pull/313`
[7]`https://hackage.haskell.org/package/mtl`
[8]`https://hackage.haskell.org/package/transformers`

system allows to combine non-parameterised effects, as shown in traditional algebraic effect systems such as `polysemy`, and scoped-parameterised effect systems as presented by Punchihewa and Wu (2021). Additionally, Punchihewa and Wu provide no abstractions that simplify implementations of parameterised algebraic effect handlers.

1.3 Contributions of this Thesis

In this thesis, we address this open problem and develop a new algebraic effect system library in Haskell, called `preff`. It allows developers to define, use and interpret algebraic effects. The algebraic effect system follows state-of-the-art research to provide a modern interface and simple usage. In addition, `preff` is capable of combining algebraic effects with scoped-parameterised effects.

We build on top of the freer monad encoding for the effect monad and extend it to the parameterised freer monad encoding. This enables user-defined algebraic effects and scoped effects. Additionally, `preff` supports scoped-parameterised effects. Scoped-parameterised effects infuse scoped effects with the ability to add type-level proofs and contracts. These proofs can be used to great advantage to significantly improve the type safety of a procedure or computation. Moreover, it enables our algebraic effect system library `preff` to implement complex control flow patterns, such as session types (Takeuchi et al., 1994). In particular, we implement a novel session type encoding in less than 120 lines of code. Finally, we provide an extension to `weave` specifically for scoped-parameterised effects.

1.4 Structure of this Thesis

In Chapter 2 we motivate the use of algebraic effect system libraries in detail. We showcase how they improve readability of code, improve maintainability and aid with testing by simulating effects. In Chapter 3, we define important type classes and data structures that are used pervasively in this thesis. Next, in Chapter 4, we discuss the implementation of `preff` in detail, argue design decisions and explain the internals of the effect system's machinery. Afterwards, we present a full picture of how to use it in practice in Chapter 5. We focus on the perspective of developers that solve real-world software issues and provide illustrative examples on how to use algebraic effects and scoped effects. Then we show how scoped-parameterised effects can be used to improve type safety and provide a novel session type encoding for our effect system. Next, we evaluate `preff` for its practical applicability in Chapter 6 by comparing the expressiveness of our novel session type encoding to existing session type libraries in Haskell. Further, we compare the run-time performance of `preff` with other state-of-the-art effect system libraries and show that `preff` is applicable in real-world scenarios. Finally, we summarise and discuss our most important findings in Chapter 7 and derive promising future research aspects.

CHAPTER 2

Motivating Algebraic Effect Systems

Algebraic effect systems can improve the readability, composability and maintainability of a software project. In this chapter, we illustrate this using a small project on processing customer data, which we extend and refine using a stepwise approach. This processing task is most naturally performed by an imperative style procedure for handling side effects.

Introducing the running example. In this example, we have to load customer data from a file, process and transform the data, followed by storing the new data in a new file. The locations for reading and storing data are known at compile-time, and we have to track how much time the processing of data takes. We provide a simple implementation of a procedure satisfying our requirements in Listing 2.1. The procedure uses the `IO` monad, which gives it direct access to a plethora of capabilities such as reading from disk, accessing the system clock, or even accessing the current network. We claim, the shown code is a viable implementation for the requirements given above, but we demonstrate the maintenance burden such an innocuous function can incur over time.

Discussing issues. The implementation from Listing 2.1 is short and concise. Nonetheless, we identify potential long-term issues.

- Having `IO` in the type signature makes it impossible to derive the possible side effects of `processCustomers`. Network access and disk access are equally possible. It increases the amount of code developers need to read and understand before the impact of using it can be understood, especially when developers are unfamiliar with the procedure.

```haskell
processCustomers :: IO ()
processCustomers = do
  customers <- readCustomersFromFile "customer.db"
  (newCustomers, execTime) <- timeAction (processData customers)
  putStrLn $ "Processing took: " ++ show execTime
  writeCustomersToFile "newCustomers.db" newCustomers
```

Listing 2.1: Simple business logic for processing data

- Testing of `processCustomers` is cumbersome. It requires setting up a fitting test infrastructure in which appropriate files exist, and removing generated artefacts after test completion.
- Logging is a cross-cutting concern, and `putStrLn` has long-term impacts on performance, readability and maintainability.
  - **Performance** Logging large amounts of data via `putStrLn` is impractical due to performance issues of `String` operations.
  - **Readability** Logging adds to the cognitive overhead required for understanding the given procedure, even though it has nothing to do with the task it fulfils.
  - **Maintainability** Writing logs to a fixed location makes it difficult to redirect logs to other locations. Adding further details requires tedious and error-prone updating.
- Last but not least, `processCustomers` cannot declare any preconditions or postconditions that are understood by the type-system. For example, we cannot communicate in the type signature of the procedure that the file `"customer.db"` must exist before running the procedure.

Admittedly, for less than ten lines of code, these concerns are negligible, since changes are trivial so far. However, the costs and concerns accumulate, the bigger the software grows, especially in the face of shifting requirements.

Refinement of the running example. We extend our example by introducing a new requirement: Callers of the procedure must be able to change the location of the two databases at run-time. To implement this requirement, we extend the procedure by two new input parameters, one for the input database, and one for the output database. Listing 2.2 shows the result of the trivial refactoring.

In the next version, we decide we need to be able to configure the procedure how we log the execution time. This is a new type of change, instead of adding parameters for plain data, such as database locations, it allows the caller to modify the *behaviour* of the procedure.

```haskell
processCustomers ::
  FilePath -> FilePath -> IO ()
processCustomers input output = do
  customers <- readCustomersFromFile input
  (newCustomers, execTime) <- timeAction (processData customers)
  putStrLn $ "Processing took: " ++ show execTime
  writeCustomersToFile output newCustomers
```

Listing 2.2: Customer processing where the database locations can be changed at run-time.

```haskell
processCustomers ::
  (String -> IO ()) -> FilePath -> FilePath -> IO ()
processCustomers logger input output = do
  customers <- readCustomersFromFile input
  (newCustomers, execTime) <- timeAction (processData customers)
  logger $ "Processing took: " ++ show execTime
  writeCustomersToFile output newCustomers
```

Listing 2.3: Customer processing with limited configurability.

We define the *behaviour* of a procedure as the sum of side effects it performs. For example, saving customer data to the database is a side effect which is part of the behaviour of a procedure. Further, a procedure is *configurable* if it allows callers to alter its behaviour. One way to alter the behaviour, is by having function parameters that implement specific side effects instead of hardcoding the side effectful monadic computation. We replace `putStrLn` with a function parameter that takes a **`String`** parameter and produces a monadic result. Thus, the logging behaviour of `processCustomers` is *configurable*. We show the result in Listing 2.3. The new requirement is implemented by introducing a function parameter to `processCustomers`.

The declaration communicates to callers of `processCustomers` that a logging function is used and file paths are used in an unspecific way. This information is not enough to know precisely what behaviour the procedure has, but it conveys more than before. Introducing a function parameter increases the flexibility and simplifies code maintenance, as the implementation can be changed without modifying `processCustomers` directly. Modifying `processCustomers` directly is error-prone, since all call-sites need to be reviewed on whether pre- and postconditions are still upheld. Further, the input parameter simplifies testing of how the procedure utilises the logger function. Taking this approach to the extreme, we can make every side effect configurable as a parameter. Then, supposedly, the procedure is maximally configurable and easier to test. In Listing 2.4 we show what this looks like.

```haskell
processCustomers ::
  (String -> IO ()) ->
  (FilePath -> IO [Customer]) ->
  (FilePath -> [Customer] -> IO ()) ->
  ([Customer] -> IO [Customer]) ->
  (IO a -> IO (a, Time)) ->
  FilePath ->
  FilePath ->
  IO ()
processCustomers logger readCustomers writeCustomers process
    timeAct input output = do
  customers <- readCustomers input
  (newCustomers, execTime) <- timeAct (process customers)
  logger $ "Processing took: " ++ show execTime
  writeCustomers output newCustomers
```

Listing 2.4: Allow configuration of every aspect of processCustomers.

```haskell
processCustomers putStrLn readCustomersFromFile
  writeCustomersToFile processData timeAction
  "customer.db" "newCustomers.db"
```

Listing 2.5: Example invocation of the procedure processCustomers from Listing 2.4.

Each side effect of processCustomers is now configurable by supplying a functional procedure at run-time. This gives us full control over the run-time behaviour of the procedure and how the business logic is actually implemented. Every functionality can be easily tested, and individual needs of callers can be tweaked without touching the code of processCustomers. However, the type signature is longer than the actual code. We also made it noticeably harder to invoke the procedure, since callers need to insert the missing implementation bits every time it is used. For illustration, we show in Listing 2.5 what calling the procedure in 2.4 could look like.

Discussing issues. Invoking processCustomers at each call site with all parameters introduces a lot of overhead for readers. However, hard-coding certain parameters defeats the initial purpose of allowing configuration of each functionality, since callers lose the gained control again. The overhead becomes more burdensome when parameters are not readily available at the call-site and need to be provided as additional parameters to the caller. This quickly leads to a blow-up of input parameters. Furthermore, information such as which parameters are used by the caller procedure and which are provided for calling additional procedures is impossible to discern from the type signature.

```haskell
class CustomerDb m where
  readCustomers :: FilePath -> m [Customer]
  writeCustomers :: FilePath -> [Customer] -> m ()

class Logger m where
  log :: String -> m ()

class Timing m where
  timeAction :: m a -> m (a, Time)

class CustomerService m where
  process :: [Customer] -> m [Customer]

processCustomers ::
  ( Logger m
  , CustomerDb m
  , Timing m
  , CustomerService m
  , Monad m
  ) =>
  FilePath ->
  FilePath ->
  m ()
processCustomers input output = do
  customers <- readCustomers input
  (newCustomers, execTime) <- timeAction (process customers)
  log $ "Processing took: " ++ show execTime
  writeCustomers output newCustomers
```

Listing 2.6: Business logic of `processCustomers` where the implementation is abstracted via type classes.

Type classes as remedy. To address the over-abundance of explicit parameters, we make use of *type classes*. We find functionality that belongs together and extract *services*. A *service* is a collection of functional procedures that are needed to fulfil a specific functionality. For example, reading and writing customer data are both used for interacting with the conceptually same database. Thus, we extract them into one service called **`CustomerDb`**. Further, we extract a **`Logger`** service, the customer processing service **`CustomerService`** and a service **`Timing`** for collecting execution time measurements. We opt to keep the database locations as regular parameters to `processCustomers`, as they are merely data. For each of these identified services, we introduce a type class and make `processCustomers` use them.

Listing 2.6 shows the implementation of `processCustomers` introducing the concept of services using type classes. Type classes avoid cluttering function parameters, while allowing to control the behaviour without modifying invocations of the procedure itself. The procedure `processCustomers` contains only business logic relevant code. Additionally, `processCustomers` does not use **`IO`** directly any more, but some monad `m` that provides all the services required to execute the procedure. This way, we lift the restriction that this procedure must be executed in the **`IO`** monad. For executing `processCustomers`, the caller may choose any monad that implements all required type classes.

The solution shown in Listing 2.6 addresses most of the initial issues criticised in the original, simple implementation of Listing 2.1:

- We no longer require the use of **`IO`**, so the monadic action `processCustomers` does not have unrestricted access to system resources.
- Testing is considerably easier, since the implementation can be chosen at run-time, by simulating *services*.

However, we have introduced an implementation issue that easily goes unnoticed in small examples. We require implementations for services, i.e. instances for type classes, and each type class can have exactly one implementation per monad. Thus, for an application, we usually introduce a type which implements the monad type class, also called the application monad. The application monad is often a wrapper over the **`IO`** monad, and provides a streamlined interface via the aforementioned type class-based services. For the concrete example procedure `processCustomers`, the application monad is the concrete instance for the monad `m` in the type signature. All services used by `processCustomers` must have an instance for the application monad type. However, if we require two different implementations for the service **`CustomerDb`**, then we also require two different application monads, since there can be only one instance, or implementation, for each service per application monad. As a consequence, when a single service shall be simulated, we have to implement all services in the application monad under test. The number of instances grows proportionally to the number of services that may have different implementations in an application.

Algebraic effect systems. In *algebraic effect systems*, we declare in the type signature, similarly to the previous approach, what services a procedure has access to. They are the *algebraic effects* of the application, describing the side effects that can occur. Procedures are then written against the abstract interface, unable to rely on any implementation details of the respective services. In other words, procedures declare which effects may be executed when running them. It is, thus, statically known which effects a procedure can use, consequentially informing users which services they need to provide. The flexibility of *algebraic effect systems* becomes obvious when trying to run a procedure: For each service, we provide the implementation, named an *algebraic effect handler* or simply *handler*.

```haskell
data Logger m a where
  Log :: String -> Logger m ()

data CustomerDb m a where
  ReadCustomers  :: FilePath -> CustomerDb m [Customer]
  WriteCustomers :: FilePath -> [Customer] -> CustomerDb m ()

data Timing m a where
  TimeAction :: m a -> Timing m (a, Time)

data CustomerService m a where
  Process :: [Customer] -> CustomerService m [Customer]

makeEffect ''Logger
makeEffect ''CustomerDb
makeEffect ''Timing
makeEffect ''CustomerService
```

Listing 2.7: Service API definitions in algebraic effect systems.

Handlers implement a single service, giving it semantics and a concrete implementation. In algebraic effect systems, services are usually referred to as effects, and we will use the two terms interchangeably. Additionally, algebraic effect handlers are functions and can be composed using function composition. At last, once all effects for a procedure are given a handler, the procedure is fully implemented.

We will now show how to write the procedure `processCustomers` using an algebraic effect system. Similar to Listing 2.6, we identify and extract the services of the application, as shown in Listing 2.7. At first glance, the definitions look very similar, especially the type signatures.

The service type classes, such as **`Logger`**, are replaced by data types, and each method is represented by a respective data constructor. Constructors are defined using generalised algebraic data types, which allows adding additional type constraints that are otherwise not trivial to express. For example, the constructors **`ReadCustomers`** and **`WriteCustomers`** produce different results, `[Customer]` and `()` respectively. Additionally, for each service, the `template-haskell`[1] function `makeEffect` is used to generate boilerplate, which we will discuss in detail in Section 4.2. Importantly for this example, it generates a function that can be used in the application code for each data constructor. It generates the functions `log`, `readCustomers`, `writeCustomers`, `timeAction` and `process`, which can then be used in the definition of `processCustomers`.

[1]https://hackage.haskell.org/package/template-haskell

```haskell
processCustomers ::
  Members '[CustomerService, TimeAction, CustomerDb, Log] f =>
  FilePath ->
  FilePath ->
  Eff f ()
processCustomers input output = do
  customers <- readCustomers input
  (newCustomers, execTime) <- timeAction (process customers)
  log $ "Processing took: " ++ show execTime
  writeCustomers output newCustomers
```

Listing 2.8: Business logic implemented using an algebraic effect system.

In Listing 2.8, we provide an implementation of `processCustomers` using an algebraic effect system. The most important difference is that instead of an arbitrary monad `m`, the computation is happening in the context of the **`Eff`** monad, the so-called *effect* monad. Services, or effects, are asserted to be part of the effect list `f` the procedure has access to. Specifying it this way does not impose any structure on `f`, meaning `f` may contain exactly these services, or many more. The order of the services is not significant either, allowing this procedure to be invoked whenever the caller also has access to the required services. Procedures are, thus, composable with other procedures using the same effect system and require no extra passing through of parameters.

Reiterating the issues we identified earlier in this section, `processCustomers` does not depend on **`IO`**, thus can be implemented completely free of side effects, if so desired. The required services are clearly specified in the type signature, documenting what the function has access to. It is impossible, accidentally or purposefully, to send network requests without requiring changes to service implementations. This increases the overall confidence that `processCustomers` does exactly what its type signature says it does. As an additional benefit, it allows developers to provide simulation implementations of various services to test specific services. Contrary to the type class based implementation given above, the implementations of services are independent of each other and can be mixed and matched as required. The concerns regarding logging are partially addressed: While it is trivial to switch out *how* text is logged, performance issues and the readability inhibition still persists. Performance considerations of logging vary greatly, depending on what information shall be logged, in what format it shall be logged and to what location logs should be sent, with different trade-offs and application requirements. Since this is out-of-scope here, we ignore potential design decisions, and focus on readability and flexibility of the logging service.

For running a procedure, each service requires an implementation or a *handler*. The handler can be seen as the server of a client-server application, while the procedure containing business logic is the client. In the running example, the client `processCustomers` sends

```haskell
runCustomerDb :: Embed IO f => Eff (CustomerDb : f) a -> Eff f a
runCustomerDb = interpret $ \case
  ReadCustomers fp ->
    embed (readCustomersFromFile fp)
  WriteCustomers fp customers ->
    embed (writeCustomersToFile fp customers)

runCustomerService :: Eff (CustomerService : f) a -> Eff f a
runCustomerService = interpret $ \case
  Process customers -> pure (processData customers)

runLogger :: Embed IO f => Eff (Logger : f) a -> Eff f a
runLogger = interpret $ \case
  Log str -> embed (putStrLn str)
```

Listing 2.9: Service implementations for Algebraic Effect Systems.

a request to the service implementation. The service handler then receives said request and performs the actual work. For instance, while the application has no dependency on `IO`, the handler might and often will have access to it. A handler can use `IO` with the `Embed IO f` constraint. In Listing 2.9, we show handler functions for the services `CustomerDb`, `Logger` and `CustomerService`, that implement the same functionality as the `IO` heavy code of Listing 2.1.

Each service defines how requests from the client are interpreted. Similar to the client code, the pattern of declaring the service dependencies continues, the handlers for `CustomerDb` and `Logger` require access to `IO`, declare this requirement via the type signature `Embed IO f`.

Assuming all handler functions are defined, we can put them all together as shown in Listing 2.10. Handlers are functions, which allow us to compose multiple handlers using `(&)` which is the reversed function application. Thus, when all effects have been handled, the final result can be extracted, using `runIO` which performs `IO` actions in the context of the chosen effect system.

Hiding implementation details. To improve readability, cross-cutting concerns, such as measuring run-time performance of code, should not be part of the application code. For most readers of `processCustomers`, neither measuring the execution time of the processing step, nor logging are relevant. Ideally, such information is hidden. Algebraic effect systems make it particularly easy to hide such implementation details by allowing to reinterpret effects in terms of other effects. For example, without having to change the handler function `runCustomerService`, we add run-time execution time measurements and logging information. In Listing 2.11, we introduce a new

```haskell
main :: IO ()
main = processCustomers "customer.db" "newCustomers.db"
  & runCustomerDb
  & runCustomerService
  & runLogger
  & runTimedAction
  & runIO
```

Listing 2.10: Tying it all together: Provide an implementation for services that are required to execute `processCustomers`.

```haskell
runTimedAndLoggedCustomerService ::
  Embed IO f =>
  Eff (CustomerService : f) a ->
  Eff (CustomerService : Logger : Timing : f) a
runTimedAndLoggedCustomerService = reinterpret3 $ \case
  Process customers -> do
    (customers, time) <- timeAction (process customers)
    log $ "Processing took: " ++ show execTime
    pure customers
```

Listing 2.11: Re-use service implementations to hide implementation details.

handler `runTimedAndLoggedCustomerService`. The handler function replaces all processing requests made by the client with more detailed requests. Note, `reinterpret3` is a utility function required to satisfy the type signature.

It is worth pointing out that the effect **`CustomerService`** is not consumed from the list of effects. Thus, it must be interpreted by a subsequent handler. The subsequent handler can still be the original handler `runCustomerService` from Listing 2.12. This indicates that it is trivial to extend existing handlers.

Summary. We have illustrated how algebraic effect systems enable developers to focus on the highest priority matters in an application by hiding implementation details where appropriate and documenting service requirements upfront in type signatures. Due to increased flexibility, they additionally make it easier to reuse existing code and test functional procedures individually with minimally increased boilerplate for effect definitions. Cross-cutting concerns can be implemented without polluting application code with unnecessary details. Moreover, algebraic effect systems can express control flow manipulating effects, such as coroutines, non-determinism and run-time exceptions.

```haskell
main :: IO ()
main = processCustomers "customer.db" "newCustomers.db"
  & runCustomerDb
  & runTimedAndLoggedCustomerService
  & runCustomerService
  & runLogger
  & runTimedAction
  & runIO

processCustomers ::
  (Members '[CustomerService, CustomerDb] f) =>
  FilePath ->
  FilePath ->
  Eff f ()
processCustomers input output = do
  customers <- readCustomers input
  newCustomers <- process customers
  writeCustomers output newCustomers
```

Listing 2.12: Execution of `processCustomers` while hiding implementation details from the business logic.

CHAPTER 3

Relevant Haskell Concepts

Haskell is a purely functional programming language. It is powered by the industrial-strength compiler GHC and comes with an extensive standard library to support developers. In this section, we present important type classes, functions and data types that are pervasively used throughout this thesis. This includes common Haskell type classes but also data types that are commonly referred to in effect system libraries.

3.1 Monad - Type Class Hierarchy

In Listing 3.1 we recall the definition of the **`Monad`** type class in Haskell as it exists at the time of this writing in the `base`[1] library version `4.19.0.0`. Adding a **`Monad`** instance for some type `m`, requires to provide instances for the type classes **`Functor`** and **`Applicative`**. We include the type class methods that are required to create an instance for the respective type classes.

The **`Monad`** type class enables developers to utilise the handy **`do`**-syntax. It allows writing seemingly sequential procedures, similar to mainstream, procedural programming languages and GHC desugars **`do`**-syntax to use methods of the **`Monad`** type class. While the exact desugaring depends on implementation details of GHC, the desugaring translates to a combination of the **`Applicative`** and **`Monad`** methods.

3.2 Parameterised Monad - Type Class Hierarchy

Parameterised monads are an extension of regular monads. The nomenclature of parameterised, indexed and graded monads seems to be inconsistent in literature. We base our naming and definitions on Orchard et al. (2020) who provide a common ground for parameterised monads (Atkey, 2009a,b). Intuitively, parameterised monads are monads

[1] `https://hackage.haskell.org/package/base`

```haskell
class Functor f where
  fmap  :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

Listing 3.1: The **Monad** type class hierarchy in Haskell. We show all method functions that are required to create an instance for the respective type class.

that are parameterised by objects representing state changes as a result of computation. We use these objects as the precondition and postcondition of monadic computations. In Haskell, there exists no standard definition of a parameterised monad.

The definitions shown in Listing 3.2 are derived from Punchihewa and Wu (2021) but intentionally name shadows instance methods from Section 3.1. Additionally, we closely mirror the **Monad** type class hierarchy. This enables us to leverage the GHC language extension **QualifiedDo**[2]. **QualifiedDo** extends GHC to permit the **do**-syntax for any module that exports the functions `(>>=)` and `pure`.

In practice, these type classes have additional methods that are omitted for the sake of simplicity. Moreover, many standard library functions have to be redefined to be compatible with the **IMonad** type class hierarchy. While there are existing libraries for parameterised monads available, none of them shadow the **Monad** type class methods. This makes them incompatible to be used with **QualifiedDo**. Thus, we provide our own implementation that is functionally equivalent to existing libraries.

As parameterised monads are a generalisation of monads, we can provide default instances for the **Monad** type class hierarchy based on the methods from the **IMonad** hierarchy.

We show these instances in Listing 3.3. The instances exploit the fact that any **IMonad** `m` permits a trivial instance of **Monad** `(m p p)`. Given the **Monad** instance, it enables developers to use **do**-syntax in procedures that have an invariant pre- and postcondition. Only the **Functor** instance permits a different pre- and postcondition as it modifies only the value of the **IFunctor** `f`.

[2] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/qualified_do.html#extension-QualifiedDo

```haskell
class IFunctor f where
  imap  :: (a -> b) -> f p q a -> f p q b

class IFunctor f => IApplicative f where
  pure  :: a -> f i i a
  (<*>) :: f i j (a -> b) -> f j r a -> f i r b

class IApplicative m => IMonad m where
  (>>=) :: m i j a -> (a -> m j k b) -> m i k b
```

Listing 3.2: The parameterised Monad type class hierarchy in Haskell. We show method functions that are required to create an instance for the respective type class.

```haskell
instance IFunctor f => Functor (f p q) where
  fmap = Ix.imap

instance IApplicative f => Applicative (f p p) where
  pure = Ix.pure
  (<*>) = Ix.(<*>)

instance IMonad m => Monad (m p p) where
  (>>=) = Ix.(>>=)
```

Listing 3.3: The parameterised type classes naturally provide instances for the standard type classes. Methods derived from the **IMonad** type class hierarchy are prefixed by **Ix** for disambiguation.

3.3 Free and Freer Monad

A *free* monad is a structure that turns some structure f into a monad without f itself having to be a monad. In Haskell, the data type **Free** is the free monad for any **Functor** f. There are various libraries available, such as `free`[3], that provide a **Free** data type and instances for well-known type classes. In this thesis, we are mostly interested in **Free** as defined by Swierstra (2008).

In Listing 3.4 we show the **Free** data type using the GADTs notation. Generalised algebraic data types have been introduced by Cheney and Hinze (2003) and are used extensively in Haskell and this thesis in particular. The **Monad** instance for **Free** f requires that `f` is an instance of **Functor** to apply the continuation `k` to it. Note that **Impure** closely resembles the `join :: Monad m => m (m a) -> m a` function.

[3]`https://hackage.haskell.org/package/free`

```haskell
data Free f a where
  Pure :: a -> Free f a
  Impure :: f (Free f a) -> Free f a

instance Functor f => Monad (Free f) where
  Pure a >>= k = k a
  Impure f >>= k = Impure (fmap (>>= k) f)
```

Listing 3.4: The **Free** data type in Haskell and its **Monad** instance for any Functor f.

```haskell
data Freer f a where
  Pure :: a -> Freer f a
  Impure :: f x -> (x -> Freer f a) -> Freer f a

instance Monad (Freer f) where
  Pure a >>= k = k a
  Impure op f >>= k = Impure op (fmap (>>= k) f)
```

Listing 3.5: The **Freer** data structure Haskell and its respective **Monad** instance.

The restriction that `f` must be an instance of **Functor** is lifted by Kiselyov and Ishii (2015) by utilising the *freer*-monad approach. In algebraic effect system libraries the **Functor** instance is only used for passing through the continuation in `f (Free f a)`. Since this is uniformly handled, we extract the continuation explicitly and put it into the **Free** structure directly, resulting in the code shown in Listing 3.5, calling the resulting monad the *freer*-monad. Thus, the constraint that `f` must be an instance of **Functor** can be dropped from the **Monad** instance for **Freer**. Similar to **Free**.**Impure**, **Freer**.**Impure** closely corresponds to the `(>>=)` operation of the **Monad** type class.

3.4 Open Union

In this thesis, we need an extensible type whose possible value range can be changed at run-time. *Union* types seem to be a good fit for this use-case (Barbanera et al., 1995), but Haskell's type system has no first-class support for union types. However, union types can be emulated using an *open union* data structure. Fundamentally, *open union* is a type-indexed co-product of operations. There are many libraries available for implementing open union data structures, e.g. `data-diverse`[4]. However, they are not always straight-forward to use and have different performance and usability trade-offs. Further, various *open union* implementations are given by Kiselyov et al. (2013), Kiselyov

[4]`https://hackage.haskell.org/package/data-diverse`

```haskell
data Op f x where
  OHere :: f x -> Op (f : effs) x
  OThere :: Op effs x -> Op (f : effs) x
```

Listing 3.6: Open union in Haskell

```haskell
interpret :: Op '[Maybe, Either Int] String -> String
interpret op = case op of
  OHere Nothing -> "No value"
  OHere (Just m) -> m
  OThere (OHere (Left l)) -> show l
  OThere (OHere (Right r)) -> r
```

Listing 3.7: Example: Extract a value from the open union data structure `Op`.

and Ishii (2015), and Wu and Schrijvers (2015) specifically to serve the needs of effect system libraries. In Listing 3.6, we show the code for the state-of-the-art *open union* structure using GADTs notation based on the prior research on algebraic effect system libraries.

The given open union `Op f x` encodes a union of all values, the types in the list `f` can represent and produces a result of type `x`. For example, a value of the type `Op '[Maybe, Either Int] String` allows to combine operations from the types `Maybe` and `Either Int` and produces a `String` value. This type represents one of four possible values:

- `OHere Nothing`
- `OHere (Just (s :: String))`
- `OThere (OHere (Left (l :: Int)))`
- `OThere (OHere (Right (r :: String)))`

We obtain the value of a variable of type `Op '[Maybe, Either Int] String` by explicit pattern matching, as shown in Listing 3.7.

CHAPTER 4

The Effect System **`preff`**

We are now ready to present our main contribution for advancing the research of algebraic effect systems, the Haskell library `preff`. Its name is a combination of **pr**otocol types and algebraic **eff**ect system, which make up the integral components of `preff`.

Algebraic effect systems allow developers to decouple implementation from intent by using effects to describe the *syntax* (i.e. a set of effect operations), and handlers to define the respective effect's *semantics* (i.e. handling of those operations). We introduce the term *protocol type* for advanced type-level proofs to encode pre- and postconditions into the effect monad. To our knowledge, `preff` is the only effect system that features user-defined algebraic effects combined with a single scoped effect that utilises powerful type-level proofs to ensure program correctness.

In the following, we first present `preff` and discuss the trade-offs of design decisions. Subsequently, we evaluate the performance in terms of execution speed and demonstrate the expressiveness of the protocol types.

4.1 The Effect Monad **`PrEff`**

In algebraic effect systems in Haskell, the effect monad is the core data structure. It provides a monadic interface, allows sequencing of effects, and is generally the first point of contact for developers. As the effect monad is pervasive in the algebraic effect system, its internals are of paramount importance, impacting run-time performance and expressiveness.

`PrEff` is the amalgamation of multiple effect monads introduced in various algebraic effect systems. Specifically, it combines the *freer*-monad approach from Kiselyov and Ishii (2015), with scoped operations as defined by Piróg et al. (2018) and the parameterised effect monad as discussed by Punchihewa and Wu (2021). The resulting parameterised freer monad supporting monadic regions is presented in Listing 4.1.

```haskell
data PrEff f s p q a where
  Value :: a -> PrEff f s p p a
  Impure ::
    Op f x ->
    IKleisli (PrEff f s) p r x a ->
    PrEff f s p r a
  ImpureP ::
    s p q x ->
    IKleisli (PrEff f s) q r x a ->
    PrEff f s p r a
  ScopedP ::
    ScopeE s (PrEff f s) p p' q' q x x' ->
    IKleisli (PrEff f s) q r  x' a ->
    PrEff f s p r a
```

Listing 4.1: Parameterised-freer monad **PrEff**.

First, we focus on the data declaration **data PrEff** f s p q a and its type variables:

- **f**: The set of algebraic effects that can be performed in **PrEff**. The type parameter f is also called the *effect signature* of **PrEff** and is represented by a list of algebraic effect types. These effects do not permit a scoping algebra, and thus, cannot have syntax with scoping semantics.
- **s**: A single scoped-parameterised effect type that has access to type-level states p and q. Contrary to effects from f, the effect s consists of two components, a base algebra and a scoped algebra, which have access to the type parameters p and q. As such, the scoped-parameterised effect is the only effect that can be used for encoding proofs at the type-level.
- **p**: The *precondition* of the parameterised monad **PrEff**. It is a *phantom* parameter, and has no run-time representation. To invoke a procedure m with the type **PrEff** f s p q a, the caller has to ensure the precondition p is met. Otherwise, the procedure m cannot be invoked. Only the effect s may assert program preconditions.
- **q**: The *postcondition* of the parameterised monad **PrEff**. Identically to p, it is a phantom parameter. After a procedure m :: **PrEff** f s p q a is invoked, the postcondition q holds and can be assumed by subsequent procedures. Only the effect s may affect a program's postconditions.
- **a**: The type of the produced value of a computation in **PrEff**.

Next, we discuss each constructor of **PrEff**.

```
data Op f x where
  OHere :: f x -> Op (f : effs) x
  OThere :: Op effs x -> Op (f : effs) x
```

Listing 4.2: Open union for the effect operations that can occur **PrEff**.

Value: The equivalent of `return`/`pure` in the **IMonad**/**IApplicative** type class. It represents a *leaf* node in the abstract syntax tree. It is a *pure* computation that may never produce any side effects. As such, the pre- and postcondition of this constructor are identical.

Impure: A node in the abstract syntax tree for algebraic operations without access to type-level information. It is a single operation of some effect `eff` that occurs in the effect signature `f` of `Op f x`. Intuitively, **Impure** corresponds to the monadic bind (>>=) operator of the **Monad** type class. All algebraic operations that can occur in **PrEff** are encoded by `Op f`, the open union of all effects in `f`. These operations are never impacted by either the pre- or postcondition of **PrEff**. We show the definition of the open union in Listing 4.2. Note, the definition is identical to the previously shown definition in Listing 3.6.

The second parameter of **Impure** is the continuation of the program, i.e. the rest of the program after the operation `Op f x` is evaluated. In the source code, we refer to the continuation via the variable `k`. The type of this continuation is:

```
k :: IKleisli (PrEff f s) q r x a
```

The name **IKleisli** is a type synonym for the Kleisli monad extended to the parameterised monad, as identified by Atkey (2009a). The type synonym simply states:

```
IKleisli m p q x a = x -> m p q a
```

However, despite the simplicity of **IKleisli**, it provides a common name for the recurring concept of the continuation of a computation in **PrEff**. It is useful for defining handler functions such as `interpretScoped`, which we discuss in Section 4.3.

The continuation `k` requires the result of the operation `Op f x` which is of type `x` . It then produces the rest of the program, which requires the precondition `q` to hold. Afterwards, the postcondition `r` holds and a value of type `a` is computed.

ImpureP: Similar to **Impure**, but operations have access to the type-level state of `p` and `q`. The main difference is that it uses `s p q x` instead of the open union `Op f x`. This allows **ImpureP** to affect the program's pre- and postcondition, enabling the developer to add complex proofs, such as session types, to the algebraic effect system. Since there is only a single effect that has access to the pre- and postcondition, **ImpureP** is not parameterised by any open union-like structure such as **Op**. The permitted operations are directly constrained via the effect type parameter `s`.

Like **Impure**, **ImpureP** closely resembles the monadic bind operation `(>>=)`, specific to algebraic operations that modify the pre- and postcondition of a program. We separate these effects from the "simple" ones to make sure, simple algebraic effects can never modify the pre- or postcondition.

ScopedP: Constructor for the scoped algebra supported by **PrEff**. It allows to capture monadic regions and provide different semantics for these regions only.

The first parameter named **ScopeE** is an *indexed data family*, which allows hiding one type parameter in the declaration of **PrEff**. It represents the scoped algebra of `s` and can be extended at compile-time with additional instances. Developers can add new *data family instances* for `s` as required. The main benefit of this slightly more complex encoding is that **PrEff** requires fewer type variables. This reduces the overall boilerplate for writing programs with `preff`, especially for programs that require no scoped-parameterised effect. In comparison, Punchihewa and Wu (2021)'s effect monad requires separate type parameters for the base and scoped algebra. Other algebraic effect system libraries also avoid the extra type parameter by tying the base and scoped algebra together in the effect definition.

The definition of **ScopeE** is shown in Listing 4.3 but it is rather concise and doesn't give a lot of insight. However, the included kind signature of **ScopeE** exposes the required type parameters that can be seen in Listing 4.1 in the usage of the **ScopedP** constructor.

Instances have to supply the following type arguments:

```
ScopeE s m p p' q' q x' x
```

The first type parameter `s` is the effect for which the scoped operation is defined. In **PrEff**, this is the scoped-parameterised effect. Since `s` is a scoped-parameterised effect, the type constructor `s` takes a pre- and postcondition, and a return type as an argument, and thus its kind signature is `forall t . t -> t -> Type -> Type`. Note, that we use `t` in order to not restrict the kind of pre- and postcondition in any way. This is consistent with the **IMonad** type class, which permits the pre- and postcondition to have any kind, not just **Type**. Furthermore, `m` is a parameterised monad action, therefore, it also has the kind signature `forall t . t -> t -> Type -> Type`, but represents an inner computation, or a scoped region. An inner computation, in the context of

```haskell
data family ScopeE s
type ScopeE :: forall t .
  (t -> t -> Type -> Type) ->
  (t -> t -> Type -> Type) ->
  t -> t -> t -> t ->
  Type -> Type ->
  Type
```

Listing 4.3: Indexed data family `ScopeE` for implementing operations with scoped monadic regions for the effect `s`. It includes a kind signature to communicate the arity of the expected type parameters.

`preff`, is a slice of a `PrEff` program, containing algebraic operations or even another scoped operation.

We recall the usage of `ScopeE` in `ScopedP`:

```haskell
ScopeE s (PrEff f s) p p' q' q x' x
```

The type parameters `p`, `q` are the pre- and postcondition of the operation identified by `ScopeE s`. In other words, the precondition `p` is required for the operation `ScopeE s` to be executed. Complementarily, `q` holds after the operation finishes execution. The inner `p'` and `q'` are the pre- and postcondition of the *inner* operation. Thus, `ScopeE s` is an operation with an *outer* and an *inner* pre- and postcondition. The scoped region, i.e. the *inner* operation, has access to the *inner* pre- and postcondition `p'` and `q'`. Since `m ~ PrEff f s`, i.e. the type variable `m` is identical to the effect monad `PrEff f s`, the scoped region is always an effectful computation, with access to the effects of `PrEff f s`. Separating the invariants for the scoped effect from the invariants of the encompassing computation, allows users to provide a translation from the initial precondition `p` to `p'`, and postcondition `q'` to `q`. Consequentially, the scoped region may produce a value of type `x'`, but the operation `s` may translate it to any type `x`. In Section 5.3, we demonstrate how the translation allows us to implement session types for dual party communication.

Further, `PrEff` is an instance of a parameterised monad. We show this by providing an implementation for `IFunctor`, `IApplicative` and `IMonad` in Listing 4.4. Each instance essentially threads the operation through the continuation of `PrEff`. Unfortunately, this can be the source of potential performance bottlenecks since unless the compiler optimises the code, operations such as `IMonad.>>=` can be expensive. In Section 6.2, we evaluate the performance compared to other algebraic effect systems in Haskell via microbenchmarks. We show that the run-time performance of `preff` is competitive with related effect system libraries and is applicable in real-world scenarios.

```
instance IFunctor (PrEff f s) where
  imap f (Value a)  = Value $ f a
  imap f (Impure  op k) = Impure  op (imap f . k)
  imap f (ImpureP op k) = ImpureP op (imap f . k)
  imap f (ScopedP op k) = ScopedP op (imap f . k)

instance IApplicative (PrEff f s) where
  pure = Value

  Value      k <*> f = imap k f
  Impure  op k <*> f = Impure  op ((<*> f) . k)
  ImpureP op k <*> f = ImpureP op ((<*> f) . k)
  ScopedP op k <*> f = ScopedP op ((<*> f) . k)

instance IMonad (PrEff f s) where
  return = pure

  Value      a >>= f = f a
  Impure  op k >>= f = Impure  op ((>>= f) . k)
  ImpureP op k >>= f = ImpureP op ((>>= f) . k)
  ScopedP op k >>= f = ScopedP op ((>>= f) . k)
```

Listing 4.4: `IFunctor`, `IApplicative`, and `IMonad` instance of `PrEff`.

For better interoperability with existing code, `PrEff` has instances for the type classes `Monad`, `Applicative` and `Functor`. We define them in terms of their parameterised versions, since they are special cases of the parameterised classes. For brevity, we omit the implementations of the standard type classes due to their simplicity.

4.2 Algebraic Effects Handlers in `preff`

User-defined algebraic effects and handlers are an essential feature of `preff`. Algebraic effect handlers have been extensively studied in the past (Plotkin and Power, 2003; Plotkin and Pretnar, 2009; Bauer and Pretnar, 2013). Especially in Haskell, different variations of algebraic effect handlers have been introduced (Kiselyov et al., 2013; Kiselyov and Ishii, 2015; Wu and Schrijvers, 2015; Wu et al., 2014; Piróg et al., 2018). In `preff`, we leverage this prior research to enable user-defined algebraic effect handlers with minimal boilerplate.

An effect is modelled as a data type with an arbitrary number of constructors. Each constructor represents an operation of the respective effect. For convenience, effects and operations are defined using GADTs. We identify the following steps for defining an

```haskell
data State s a where
  Put :: s -> State s ()
  Get :: State s s

get ::
  Member (State e) f => PrEff f s p p e
get =
  send Get

put ::
  Member (State e) f => e -> PrEff f s p p ()
put s =
  send (Put s)

runState :: ScopedEffect s =>
  e ->
  PrEff (State e : f) s p q a ->
  PrEff f s p q (e, a)
runState initial = interpretStateful initial $ \s -> \case
  Get -> pure (s, s)
  Put newS -> pure (newS, ())
```

Listing 4.5: Define the effect **State**, embed its operations and implement its respective handler function runState.

algebraic effect in preff.

- Define the effect and its operations using GADTs.
- Embed the operations, such that they can be used in a monadic **PrEff** computation.
- Define an algebraic handler function for the effect.

In Listing 4.5, we demonstrate the necessary steps to define the **State** effect. Additionally, we include the handler runState that provides the implementation of the **State** effect.

Operations of an effect, such as **Put** and **Get**, need to be embedded into the monadic context of **PrEff**. In particular, given an open union **Op** f, where f is the effect signature of **PrEff**, the operation op :: eff needs to be embedded into **Op** f depending on the index of eff in the effect signature f. In other words, the exact embedding depends on the effect signature f and its order of effects during execution. Since the order of effects is unknown when defining get and put, we use send which takes care of embedding an operation appropriately. As we can see in Listing 4.6, send essentially creates a

```haskell
send :: Member eff f => eff a -> PrEff f s p p a
send f = Impure (inj f) (\x -> pure x)
```

Listing 4.6: Injection of algebraic effects into **PrEff**.

```haskell
class Member eff f where
  inj :: eff a -> Op f a

instance {-# OVERLAPPING #-} Member e (e ': effs) where
  inj :: f a -> Op (f : effs) a
  inj e = OHere e

instance Member eff f => Member eff (e ': f) where
  inj = OThere . inj
```

Listing 4.7: **Member** type class for injection into the open union **Op**.

node of the **PrEff** monad and injects the operation into the open union via `inj`. The function name `send` is reminiscent of the analogy that algebraic effect systems consist of a decentralised authority that responds to requests. Thus, a **PrEff** program *sends* a request to the decentralised authority, i.e. a handler, which in turn assigns semantics to the syntax of **State**, e.g. **Get** and **Put**. To inject an operation into **Op** `f` we use the function `inj` from the type class **Member**. This type class represents a compile-time promise that we can inject the operation into the open union. Further, `send` sets the continuation of **Impure** to the identity operation. Executing the continuation produces the result of the operation. However, this continuation is extended by sequencing operations via `(>>=)`, resulting in an AST of the application program.

The type class **Member**, first introduced by Kiselyov and Ishii (2015), has two purposes: Inject an operation into the open union and declare in the type signature of a procedure that certain effect operations may be used. We show the definition of the **Member** type class in Listing 4.7, omitting the final instance which improves the error message if a **Member** constraint cannot be satisfied.

Next we look at algebraic handlers in `preff`. The syntax of an algebraic effect system is defined by the signature of its effects. However, the semantics is decoupled from the syntax and is defined via *algebraic effect handler* functions. Programs using algebraic effect systems consist of multiple handler functions that can be sequentially composed into a decentralised authority. This decentralised authority defines the program's semantics. Handlers translate the algebraic operations of an effect to an executable program. Usually, a handler handles a single effect of the effect signature `f`. Once an effect is handled, it is removed from the effect signature `f`, and no operations of the effect remain in the AST

```
runState' ::
  (ScopedEffect s) =>
  e ->
  PrEff (State e : f) s ps qs a ->
  PrEff f s ps qs (e, a)
runState' initial = \case
  Value x -> pure (initial, x)
  Impure (OHere op) k -> case op of
    Put s -> runState' s (k ())
    Get -> runState' initial (k initial)
  Impure (OThere op) k ->
    Impure  op (\x -> runState' initial (k x))
  ImpureP op k ->
    ImpureP op (\x -> runState' initial (k x))
```

Listing 4.8: Inductive recursion over the program structure of **PrEff** to implement the handler `runState`. Only the highlighted lines specify the semantics of the **State** effect.

of the **PrEff** computation. Once all effects have been handled, i.e. the effect signature `f` is empty, only a single computation remains. This computation is then evaluated to a single value and may produce side effects, depending on the final monadic context.

While `preff` provides many utility functions for defining handlers such as `interpret` and `interpretStateful`, it is possible to directly traverse the AST of a **PrEff** program. For example, it is semantically equivalent to implement `runState` like `runState'` as shown in Listing 4.8. However, most of the inductive traversal is boilerplate that is irrelevant to the semantics of `runState'`. The utility functions for implementing handlers, such as `interpretStateful`, simply abstract over these patterns. Therefore, while all effect handlers in `preff` can be written using inductive recursion, the users of `preff` are encouraged to use the utility functions provided by `preff` for defining algebraic effect handlers whenever possible. In addition to being much simpler to implement and understand, it also allows `preff` to apply optimisations during AST traversals.

So far, we have omitted the constructor **ScopedP** of **PrEff**. Handling the operations within a **ScopedP** node is more complex, since the structure of the operation is opaque to `runState'`. Thus, we need to *weave* effect handlers through scoping operations.

Weaving of Handlers through Scoped Algebras

Handling of an algebraic effect requires to also handle the operations within the scoped regions of **ScopedP**. However, since the scoped-parameterised effect's operations are opaque to handlers of other effects, this requires special care. A handler of an effect has

```
class ScopedEffect s where
  weave :: Functor ctx =>
    ctx () ->
    (forall r u v . ctx (m u v r) -> n u v (ctx r)) ->
    ScopeE s m p p' q' q x x' ->
    ScopeE s n p p' q' q (ctx x) (ctx x')
```

Listing 4.9: Recapitulate: Definition of the type class `ScopedEffect` which enables weaving of algebraic effect handlers though scoped effects.

to be weaved through the scoped operation together with the context of the handler. In Listing 4.9, we introduce the type class, `ScopedEffect`, which represents the class of scoped-parameterised algebras that can thread handlers through their scoped monadic regions. It is a direct adaptation of the `handle` function from Wu et al. (2014). The function `weave` takes as a first argument the initial context and a transformation function. Then this transformation function is applied to each inner region of operations of `s` and adds the context to the resulting output. Within this transformation, we apply effect handlers of other effects, ensuring all operations of an effect are properly handled. All scoped algebras that allow weaving handlers need to implement an instance of `ScopedEffect`. In Section 4.3, we provide concrete examples for scoped-parameterised effects that implement the `ScopedEffect` type class.

Given the definition of `ScopedEffect`, we can now complete the `runState'` handler in Listing 4.10. The idea is to weave the effect handler through the inner computations of a scoped operation node. In particular, it is important to weave intermediate state between the scoped region and the continuation of `ScopedP`. Otherwise, changes to `e`, the mutable variable of `State`, would be lost. In the usage of `weave`, the initial context is the current state and a context token `(e, ())`, i.e. `ctx () ~ (e, ())`. This context is then woven via a transformation through the scoped operation of `ScopeE`. The transformation applies the handler function for the `State` effect to any possible scoped operations in `ScopeE s`. Finally, the handler is invoked on the continuation of `ScopedP` with the state obtained from the `weave` operation.

As we have demonstrated in Listing 4.8 and Listing 4.10, it is feasible, and not too difficult, to write algebraic effect handlers without any utility function such as `interpret` or `interpretStateful`. However, these utility functions abstract over the most common use-cases and should be used whenever possible. We conjecture, that it is possible to implement all algebraic effects using the utility functions provided by `preff`.

4.3 Scoped-Parameterised Effect Handlers in preff

The implementation is based on the contributions of Piróg et al. (2018) and Punchihewa and Wu (2021). As such, the constructor `ScopedP` encodes the scoped algebra in

```haskell
runState' ::
  (ScopedEffect s) =>
  e ->
  PrEff (State e : f) s p q a ->
  PrEff f s p q (e, a)
runState' s = \case
  ...
  ScopedP op k ->
    ScopedP
      ( weave
          (s, ())
          ( \(s', inner) -> Ix.do
              (x, newS) <- runState' s' inner
              pure (x, newS)
          )
          op
      )
      (\(s', a) -> runState' s' (k a))
```

Listing 4.10: Completed handler `runState'` for `State e` from Listing 4.8.

`PrEff` and is extensible via the data family `ScopeE`. Similarly to algebraic effects, scoped-parameterised effects consist of the following components:

- The base algebra component of the scoped-parameterised effect.
- The scoped algebra component of the scoped-parameterised effect.
- An embed function for each operation of both components such that the operations can be used in monadic `PrEff` computations.
- The effect handler for the scoped-parameterised effect.

We illustrate the scoped algebra on the scoped-parameterised effect `StateP`. All operations are suffixed with $\star$`P`, which stands for *parameterised*, to disambiguate its operations from the non-parameterised `State` effect. Similar to `State`, `StateP` maintains a single mutable variable. Contrarily, the type of this mutable variable may change during program execution. We show how we implement this in `preff` and discuss the relevant details. However, we skip over detailed explanations of type variables that have already been introduced in Section 5.2.1.

The effect `StateP` supports two regular algebraic operations, `PutP` and `GetP`. These are semantically analogous to common state effects or monads. However, `PutP` can

```haskell
data StateP p q a where
  PutP :: x -> StateP p x ()
  GetP :: StateP p p p

data instance ScopeE StateP m p p' q' q x x' where
  ZoomP ::
    (p -> p') ->
    (q' -> q) ->
    m p' q' x ->
    ScopeE StateP m p p' q' q x x

putP ::
  p ->
  PrEff effs StateP q p ()
putP p = sendP (PutP p)

getP ::
  PrEff effs StateP p p p
getP = sendP GetP

zoomP ::
  (p -> p') ->
  (q' -> q) ->
  PrEff effs StateP p' q' a ->
  PrEff effs StateP p q a
zoomP f restore act =
  sendScoped (ZoomP f restore act)
```

Listing 4.11: Operations of the parameterised **StateP** effect.

update both the value and the type of its mutable variable. Further, we introduce the scoped operation **ZoomP**. It allows to “zoom in” on a datatype and the scoped region only has access to the changed mutable variables. After the scoped region has finished execution, the resulting state may be fed back into the original mutable variable. It is clear, that the effect **StateP** requires access to the pre- and postcondition of **PrEff**.

Effects that require access to the type-level information can be defined similarly to *simple* effects, but often require more type annotations. To differentiate such effects from *simple* effects, we call effects that can access the type variables p and q *parameterised* effects. Parameterised effects also permit an algebraic interpretation, just like simple effects, and are expressed via GADT syntax. Definition and usage of a parameterised **StateP** effect, which can change the type of its mutable value, are shown in Listing 4.11.

We briefly discuss the definition of the base component `StateP` which is free of scoped operations.

- `PutP :: x -> StateP p x ()`: The update operation of the mutable variable. It expects some value of type `x` and represents an operation that produces the unit result `()`. The precondition is any `p`, which communicates that this operation can always be invoked, no matter what the type of the mutable variable is. After the operation `PutP` is performed, the type of the mutable variable has changed to `x`.
- `GetP :: StateP p p p`: Get the current value of the mutable variable. Similarly to `PutP`, this operation can always be used since its precondition is a simple type variable `p`. Moreover, it leaves the type of the mutable variable invariant, thus, the postcondition stays the same. The only difference is that `GetP` produces a value of type `p`.

To define scoped-parameterised effects, we have to add a data family instance for the data family `ScopeE`. This provides two advantages:

- It enables the developer to change the scoped-operations of a particular scoped-parameterised effect at compile-time.
- It avoids an additional type parameter to `PrEff`. Some algebraic effect systems have separate types for the base- and scoped algebra of an effect, such as Punchihewa and Wu (2021).

The type declaration of `data instance ScopeE StateP` declares the scoped operations of the `StateP` effect. It carries a parameterised monad `m` type parameter, which is used to describe the monadic regions the operation scopes over. In `preff`, it holds that `m ~ PrEff f s`, where `f` is the effect signature, i.e. a list of simple effects, and `s` is the scoped-parameterised effect itself, thus `s ~ StateP`.

We define a single scoped-parameterised operation `ZoomP`. `ZoomP` allows users to temporarily change the type of the value it carries from `p` to `p'`, for the duration of executing `m p' q' x`. After the scoped operation ends, the state value the operation `m p' q' x` produces, may be changed, e.g. incorporated into the original value. Since the postcondition is `q'`, the type of the mutable variable is also `q'`.

Embedding the operations of `StateP` for usage in `PrEff` is accomplished using the utility functions `sendP` and `sendScoped`. These utility functions are analogous to `send` and we show the implementation in Listing 4.12. They essentially create a `PrEff` node and initialise continuation, i.e. the second parameter to both `ImpureP` and `ScopedP` simply produces the result of their respective operation. Differently to `send`, the operation, e.g. `PutP`, can be used directly in the definition since there is only a single scoped-parameterised effect `s`. Thus, there is no need for a `Member` constraint. Especially in

```haskell
sendP ::
  s p q a ->
  PrEff f s p q a
sendP s =
  ImpureP s (\x -> pure x)

sendScoped ::
  ScopeE s (PrEff f s) p p' q' q x' x ->
  PrEff f s p q x
sendScoped scopedOp =
  ScopedP scopedOp (\x -> pure x)
```

Listing 4.12: Injection of algebraic effects into **PrEff**.

the type signature of `sendScoped`, it is highlighted that the monad parameter `m`, the **ScopeE** definition from above, is always **PrEff** `f s`.

Since the **StateP** effect has now been defined, we continue with handling a scoped-parameterised effect in `preff`. Similar to Section 4.2, algebraic effect handlers can be defined in two ways: inductively recurse over the constructors of **PrEff**, or use one of various utility functions. As before, we present both approaches, starting with the manual recursion over **PrEff** in Listing 4.13. The handler consists mostly of boilerplate to recurse over the structure of **PrEff**. While the handler for the operations **GetP** and **PutP** is familiar by now, handling the scoped operation is more involved. The handler needs to be applied to the inner action `act`, and its intermediate state needs to be modified to appropriately "zoom" the state. After the scoped effect of the inner region has been handled, we continue with the continuation of **ScopedP**, i.e. the rest of the program.

The boilerplate from handling the scoped effect using inductive recursion can be avoided by using one of the utility functions provided by `preff`. For defining algebraic effect handlers that require intermediate state, `interpretStatefulScoped` is provided. As we can see in Listing 4.14, this enables us to focus on the semantics of the effect, instead of having to worry about passing the correct arguments. The first argument to `interpretStatefulScoped` handles the base algebra of **StateP** which is identical to the effect handler `runState` as seen in Listing 4.5. The second argument is a function which handles the scoped algebra of **StateP**, e.g. the operation **ZoomP**. This function parameter takes three arguments: `run`, `continuation` and `state`. First, the `run` argument allows the developer to control in which order scoped regions of the operation are evaluated, or even evaluated at all. The function `run` is a reference to the handler itself, i.e. `runStateP`, allowing the developer to handle the **StateP** effect in scoped regions. Next, the argument `continuation` represents the rest of the program which the developer needs to run to continue program execution after the effect operation

```haskell
runStateP' ::
  p ->
  PrEff f StateP p q a ->
  PrEff f IVoid () () (q, a)
runStateP' p = \case
  Value x -> pure (p, x)
  Impure op k ->
    Impure op (\x -> runStateP' p (k x))
  ImpureP op k -> case op of
    PutP x -> runStateP' x (k ())
    GetP -> runStateP' p (k p)
  ScopedP op k -> case op of
    ZoomP f restore act -> do
      (q', a) <- runStateP' (f p) act
      runStateP' (restore q') (k a)
```

Listing 4.13: Inductively recurse over **PrEff** to define the semantics of the **StateP** effect. Only highlighted lines of code implement the semantics of the effect, whereas the rest is required boilerplate.

ZoomP has been handled. Finally, the argument `state` is the current value of the mutable variable.

Note, the utility function `interpretStatefulScopedH` simplifies the example further, as it exploits the fact that most handlers continue normal program execution after the scoped regions have been handled. Thus, scoped algebra handling of the utility function `interpretStatefulScopedH` has no access to `continuation`, and only controls handling of scoped regions.

Similar to simple effects, we encourage users of `preff` to prefer utility functions over manually recursing over **PrEff**. However, we acknowledge that the use-case and capabilities of the scoped-parameterised effects sometimes warrant writing an algebraic effect handler without utility functions. In Section 5.3, we provide such a use-case when evaluating the expressiveness of `preff`.

4.3.1 Weaving of Handlers through Scoped Algebras

We return to the type class **ScopedEffect** which we introduced in Section 4.2, Listing 4.9. The type class is required to implement weaving of algebraic effect handlers through the inner regions of scoped effects. As such, all scoped-parameterised effects should have an instance for **ScopedEffect**. Creating an instance for **ScopedEffect** is usually simpler than the intimidating type signature might suggest.

```haskell
runStateP ::
  p ->
  PrEff f StateP p q a ->
  PrEff f IVoid () () (a, q)
runStateP =
  interpretStatefulScoped
    do \state -> \case
        PutP newState -> pure (newState, ())
        GetP -> pure (state, state)

    do \continuation run state -> \case
        ZoomP f restore act -> do
          (x, newState) <- run (f state) act
          run (restore newState) $ continuation x
```

Listing 4.14: Define the effect handler of the **StateP** effect. The effect system library `preff` provides the utility function `interpretStatefulScoped` which simplifies the implementation of the effect handler.

```haskell
class ScopedEffect s where
  weave :: Functor ctx =>
    ctx () ->
    (forall r u v . ctx (m u v r) -> n u v (ctx r)) ->
    ScopeE s m p p' q' q x x' ->
    ScopeE s n p p' q' q (ctx x) (ctx x')
```

Listing 4.15: Recapitulate: Definition of the type class **ScopedEffect** which enables weaving algebraic effect handlers though scoped effects.

For convenience of the reader, we recall the definition of the type class **ScopedEffect** in Listing 4.15.

We illustrate how to add an instance for **ScopedEffect** on the case-study of the parameterised state effect **StateP** with its previously defined scoped algebra **ZoomP** in Listing 4.16. The operation `op` is transformed to contain the necessary context of effect handlers. Then the transformation function, named `nt`, is applied to the scoped region. This transformation, in `preff`, is an effect handler for some effect `f` in the effect signature of `PrEff f s`. Thus, it makes sure that the effect's operations are handled in the scoped region. The context token `ctx ()` allows us to keep track of intermediate results, as we have shown in Listing 4.10.

Not all scoped-parameterised effects can implement the rigid interface of **ScopedEffect**.

```haskell
instance ScopedEffect StateP where
  weave ctx nt (ZoomP f restore act) =
    ZoomP f restore (nt $ act <$ ctx)
```

Listing 4.16: **ScopedEffect** instance for **StateP**. The transformation function argument is applied to the scoped monadic region in **ZoomP**.

As a consequence, it limits the order of applying effect handlers and forces the user to always handle the scoped-parameterised effect first. Thus, such a scoped-parameterised effect is less flexible, but still usable.

4.3.2 Strengths and Limitations

Our algebraic effect system library `preff` allows developers to combine the benefits of traditional algebraic effect system libraries with the improved type safety of parameterised monads. Algebraic effects in `preff` are convenient to define and use, allowing developers to opt-in to more advanced type-level proofs when beneficial. Scoped-parameterised effects in `preff` are used to implement complex control flow patterns, such as coroutines, iterators or even session types. The limitation that `preff` permits only a single scoped-parameterised effect is necessary to ensure soundness. If there was more than one scoped-parameterised effect, it would allow subverting the type safety provided by the pre- and postcondition in **PrEff** due to unexpected control flow changes. Another potential design could implement a central authority that coordinates the overall control flow, however, this would deteriorate the composability of the algebraic effect system. Thus, it is a conscious tradeoff of `preff` to support only a single scoped-parameterised effect to ensure soundness and composability.

CHAPTER 5

A Tour of **preff**

User-defined algebraic effects make up the core of any algebraic effect system. We demonstrate our algebraic effect system by implementing the algebraic effects **`Reader`**, **`Writer`** and **`State`**. These effects correspond closely to their monadic counterparts, and are composable similar to the monad transformers **`ReaderT`**, **`WriterT`** and **`StateT`**. Afterwards, we show usage of our algebraic effect system on the example from Section 2 by writing the procedure `processCustomers` in `preff`. We define the effects that we require for `processCustomers`, and show how to use **`IO`** procedures. Next, we introduce scoped-parameterised effects in `preff` by applying scoped effects to `processCustomers` to improve type safety. Finally, we use scoped-parameterised effects to introduce a novel encoding of binary session types to demonstrate the expressiveness of `preff`.

5.1 Algebraic Effects and Handlers

We require three steps to define and use an effect in `preff`:

1. Define the algebraic effect using GADT where each constructor presents an effect operation.
2. Embed each operation such that it can be used in **`PrEff`**, the effect monad of `preff`.
3. Provide a handler function that gives the effect meaning, i.e. an implementation.

We demonstrate these steps on well-known effects, followed by implementing the running example `processCustomers` in `preff`.

```haskell
data Reader r a where
  Ask :: Reader r r

data Writer w a where
  Tell :: w -> Writer w ()

data State e a where
  Put :: e -> State e ()
  Get :: State e e
```

Listing 5.1: The `Reader`, `Writer` and `State` effect in `PrEff`.

5.1.1 Defining the Effects `Reader`, `Writer` and `State`

Algebraic effects allow implementing monads that are composable in a similar way to their respective monad transformer counterparts. We illustrate the capabilities of `preff` by implementing effects for the monad transformers, `ReaderT`, `WriterT` and `StateT`. These effects are frequently used in most effect systems, due to their general usefulness as building blocks for other effects.

First, we define the effects and the respective operations they can perform: `ReaderT` provides the operation `ask`, which allows the developer to ask for the value of a read-only variable, `WriterT` provides `tell` to collect values, and `StateT` provides `put` and `get` operations for updating and retrieving the value of a mutable variable. Since the effects mirror the monad transformers, the algebraic effect definitions should have the same operations.

We define the effects `Reader`, `Writer` and `State` and provide their algebraic effect definition using GADTs in Listing 5.1. Each effect must be parameterised by at least one type parameter for the output type, named `a` in these examples. They may have arbitrarily many additional type parameters:

- `Reader` can hold a read-only variable of type `r`.
- `Writer` writes values of type `w` that can be read by a consumer later.
- `State` can hold a mutable variable of type `e`.

Each constructor of the data types describes one operation of the respective effect. Using GADTs allows each constructor to have a different result type. For example, `Put` takes a value of type `e` and then produces the unit result, as indicated by `State e ()`. The corresponding `Get` operation takes no parameters and produces a value of the same type, namely `e`.

The effect monad **PrEff** is notably different to comparable effect monads of most other algebraic effect systems in Haskell. In most other algebraic effect systems, the effect monad is parameterised by two type parameters. **PrEff** has five type parameters, usually denoted as **PrEff** f s p q a:

- **f** is the effect list of **PrEff**, i.e. the list of all effects that can be performed in a procedure. We also refer to f as the *effect signature* of **PrEff**.
- **s** is the scoped effect of **PrEff**. The scoped effect is discussed in detail in Section 5.2.
- **p** and **q** are the pre- and postcondition which are required for scoped effects. However, they are not relevant for algebraic effects.
- **a** is the result type of a **PrEff** computation.

An effect's operation is simply data, expressed as a data constructor. To invoke an effect operation in the effect monad **PrEff**, the operation needs to be wrapped or embedded into the context of the effect monad. We name this process *wrapping* or *embedding* an effect in **PrEff**. Therefore, we embed the operation into **PrEff** by implementing an embedding function, e.g. the effect operation **Ask** :: **Reader** r r requires an embedding function ask :: **PrEff** f s p p r. Conceptually, we want to inform **PrEff** that a certain operation will be performed in the program.

To embed operations to be used in **PrEff**, preff provides the utility function **send**. The **send** functions allow us to "send" the operation to **PrEff**, where it is later handled by a handler function. It is only allowed to "send" an operation of the effect eff if eff is part of the effect signature. However, since an embedding function, or monadic procedure, shouldn't specify the exact instantiation of f, we constrain the effect list of **PrEff** via the type class constraint **Member** eff f. Intuitively, this constraint expresses that eff is an element in the effect list f. Further, **Member** eff f expresses, that a function can only be invoked if the effect eff is part of the effect list f. Thus, ask has the type class constraint **Member** (**Reader** r) f, tell has **Member** (**Writer** w) f and put and get have both **Member** (**State** e) f. Additionally, the type of the produced value of the embedding function corresponds exactly to the result type of each operation respectively. For example, **Ask** produces a value of type r and thus, ask has the type **PrEff** f s p p r while **Tell** produces unit, reflected by **PrEff** f s p p (). We show the full code for wrapping the operations of each effect in Listing 5.2. As we can see, there is a close correspondence between the effect operation, identified by a constructor, and the embedding function that allows us to perform the effect in **PrEff**. By convention, the embedding function for an effect operation is the lowercased name of the operation, for example for the effect operation **Ask**, we name the embedding function ask. It is trivial to automatically generate the embedding functions that follow the convention via template-haskell[1], as we have shown in Listing 2.7. After embedding the operations in **PrEff**, we can use the freshly defined algebraic effects.

[1]https://hackage.haskell.org/package/template-haskell

```
ask :: Member (Reader r) f => PrEff f s p p r
ask = send Ask

tell :: Member (Writer w) f => PrEff f s p p ()
tell x = send (Tell x)

put :: Member (State e) f => e -> PrEff f s p p ()
put x = send (Put x)

get :: Member (State e) f => PrEff f s p p e
get = send Get
```

Listing 5.2: Example: Embed the operations of the effects **Reader**, **Writer** and **State** in **PrEff**. This is required to use the operations in the effect monad. Usually, these functions are automatically generated at compile-time.

```
incrementByTwo :: Member (State Int) f => PrEff f s p p String
incrementByTwo = do
  i <- get @Int
  put (i + 2)
  pure $ show i
```

Listing 5.3: Example: Increment the value of a **State** effect by two.

To use an effect in a procedure, the effect must be in the effect list `f`. As mentioned before, this constraint is expressed by `Member eff f`. In Listing 5.3, we present a simple procedure that requires a `State Int` effect and increases its mutable variable by tw, then produces a **String** from the original integer value.

Algebraic effect systems must offer the possibility of writing handlers. A handler provides an *interpretation* of an effect. There may be multiple interpretations for a single effect. The library `preff` comes with a lot of convenience functions to help users define effect handlers. Algebraic effect handlers are often defined using only the provided abstractions, simplifying the implementation. Moreover, users do not need any knowledge of the internal structure of **PrEff** to be able to implement effect handlers. In Listing 5.4, we present the handlers for the effects **Reader**, **Writer** and **State**. We showcase the utility functions:

- **interpret**: An essential helper that allows the user to define the semantics of an effect in **PrEff**. Users simply provide the implementation for the operations. The implementation can utilise other effects of **PrEff**, i.e. the effect is interpreted

```haskell
runReader :: ScopedEffect s =>
  r ->
  PrEff (Reader r : f) s p q a ->
  PrEff f s p q a
runReader e = interpret $ \case
  Ask -> pure e

runWriterViaMonoid :: (ScopedEffect s, Monoid w) =>
  PrEff (Writer w : f) s p q a ->
  PrEff f s p q (w, a)
runWriterViaMonoid = interpretStateful mempty $ \s -> \case
  Tell w -> pure (s  w, ())

runState :: ScopedEffect s =>
  e ->
  PrEff (State e : f) s p q a ->
  PrEff f s p q (e, a)
runState initial = interpretStateful initial $ \s -> \case
  Get -> pure (s, s)
  Put newS -> pure (newS, ())
```

Listing 5.4: Example: Writing handlers for **Reader**, **Writer** and **State** effects.

in the context of **PrEff**. Especially convenient for effects that do not require any intermediate state, such as **IO** operations or the handler `runReader`.

- **interpretStateful**: Like `interpret`, effects are interpreted in **PrEff**. Thus, the handler can also depend on other effects, simplifying the implementation. In addition, `interpretStateful` carries a value that can be modified between each two operation call. As such, this helper aids in implementing effect handlers with intermediate state, such as the `runState` handler.

Since `runReader` requires no intermediate state, its handler can be implemented using `interpret`. Naturally, `runState` requires intermediate state, and thus, it uses `interpretStateful` to handle the modification of mutable variables. We implement `runWriterViaMonoid` using `interpretStateful` for the sake of simplicity. The state variable is used as an append-only variable. In Section 4.2, we discuss the details of handler functions, and how to write them without any utility functions.

The type signatures of the handler functions are mostly straightforward, explained below for `runReader`:

- The handler takes an effectful **PrEff** computation, where the first element of the effect signature is **Reader** r, specified by **PrEff** (**Reader** r : f).
- It produces a new computation where all occurrences of **Reader** r have been handled, resulting in **PrEff** f.
- The types of the pre- and postcondition may be different, and thus they have two distinct type variables p and q. However, handling the **Reader** effect may not affect the pre- and postcondition. The resulting computation must have the same type for the pre- and postconditions. Note, algebraic effects as defined so far can never affect the value of the type of the pre- and postcondition.
- At last, **ScopedEffect** s is necessary boilerplate to enable more flexible handling of scoped effects. The details of this type class are discussed in Section 4.3.

By now, we have defined each effect, e.g. **Reader**, their respective operations, e.g. **Ask**, embedded the effects for use in **PrEff**, e.g. ask, and implemented the respective algebraic effect handler function, e.g. runReader. This suffices for usage in preff.

5.1.2 The Running Example: processCustomers in preff

After demonstrating how to define and handle algebraic effects in preff, we proceed with migrating processCustomers to the algebraic effect system preff. As a reminder, we ended the discussion of the procedure processCustomers with the code shown in Listing 5.5. The procedure processCustomers was designed according to a generic algebraic effect system and is valid in most common algebraic effect systems, except for the name of the effect monad. In that spirit, to make the procedure compatible with preff, the only required modification is changing the effect monad to **PrEff** f s p p (), as we highlight in Listing 5.6. Naturally, the effect monad has a different name and we need to add a type variable s for a scoped effect. In this procedure, the pre- and postconditions must not change. Thus, we choose the same type variable p for both pre- and postcondition. However, this will change in Section 5.2. The return type is identical to before, no result is produced and thus we choose ().

In processCustomers, we use the effects **CustomerService** and **CustomerDb** for the implementation. Identically to the effects defined above, we have to define the effects, their respective operations, embed them in a monadic **PrEff** computation and implement algebraic effect handlers. We define the operations using GADTs and handlers for the effects **CustomerService** and **CustomerDb** in Listing 5.7. For brevity, we omit the definition of process, readCustomers and writeCustomers and assume the existence of fitting functions processData, readCustomersIO and writeCustomersIO.

The algebraic effects for **CustomerService** and **CustomerDb** themselves are straightforward. Definitions of the handler functions are mostly familiar, using the utility

```haskell
processCustomers ::
  Members '[CustomerService, CustomerDb] f =>
  FilePath ->
  FilePath ->
  Eff f ()
processCustomers input output = do
  customers <- readCustomers input
  newCustomers <- process customers
  writeCustomers output newCustomers
```

Listing 5.5: Example: Customer processing using a generic algebraic effect system.

```haskell
processCustomers ::
  Members '[CustomerService, CustomerDb] f =>
  FilePath ->
  FilePath ->
  PrEff f s p p ()
processCustomers input output = do
  customers <- readCustomers input
  newCustomers <- process customers
  writeCustomers output newCustomers
```

Listing 5.6: Example: Customer processing using the algebraic effect system `preff`. Only the effect monad needs to be changed.

function `interpret` for easy handling of the operations. A new feature is the constraint `Member (Embed IO) f` in `runCustomerDbIO` which allows handlers direct access to the `IO` monad. The effect `Embed m` is an effect that allows wrapping any monad `m` to be used in `PrEff`, for example `IO`. Since the handlers use pre-defined `IO` procedures, this definition is quite succinct. However, since there may be more than one distinct handler for an effect, we can provide alternative implementations that require no `IO` without modifying `processCustomers` at all. For example, a completely pure implementation leverages the `State` effect in Listing 5.8. It uses an associative variable, that maps a `FilePath` to `[Customer]`. The operation `ReadCustomers` looks up the associated customers, while `WriteCustomers` saves new customers to the mutable variable.

5.2 Scoped-Parameterised Effects and Handlers

By now, we have implemented `processCustomers` using our algebraic effect system library `preff`. While this allows developers to benefit from the flexibility and composability of algebraic effect systems, there are classes of errors that remain unaddressed, not

```haskell
data CustomerService a where
  Process :: [Customer] -> CustomerService [Customer]

data CustomerDb a where
  ReadCustomers :: FilePath -> CustomerDb [Customer]
  WriteCustomers :: FilePath -> [Customer] -> CustomerDb ()

runCustomerService ::
  ScopedEffect s =>
  PrEff (CustomerService : f) s p q x ->
  PrEff f s p q x
runCustomerService = interpret $ \case
  Process customers ->
    pure $ processData customers

runCustomerDbIO ::
  (Member (Embed IO) f, ScopedEffect s) =>
  PrEff (CustomerDb : f) s p q x ->
  PrEff f s p q x
runCustomerDbIO = interpret $ \case
  ReadCustomers fp ->
    embed $ readCustomersIO fp
  WriteCustomers fp customers ->
    embed $ writeCustomersIO fp customers
```

Listing 5.7: Example: Effect definitions and handlers for `processCustomers`.

```haskell
runCustomerDbViaState ::
  (Member (State (Map FilePath [Customer])) f, ScopedEffect s) =>
  PrEff (CustomerDb : f) s p q x ->
  PrEff f s p q x
runCustomerDbViaState = interpret $ \case
  ReadCustomers fp -> do
    customerMap <- get
    pure (customerMap ! fp)

  WriteCustomers fp customers -> do
    customerMap <- get
    put (insert fp customers customerMap)
```

Listing 5.8: Example: A pure implementation of the effect handler of **`CustomerDb`**.

fully utilising the capabilities of `preff`. For example, if a programming error occurs and the input database of `processCustomers` does not exist, then the procedure crashes at run-time. In other words, programmer assumption and preconditions are verified by developers, not by the compiler, and can thus be violated in a non-trivial fashion. Our algebraic effect system library excludes these classes of programming errors via *scoped-parameterised* effects. Scoped-parameterised effects consist of two components referred to as the *base* algebra and the *scoped* algebra, formalised by Piróg et al. (2018). Additionally, these effects are parameterised by a pre- and postcondition that can be used to encode proofs at the type-level. To define a scoped-parameterised effect in `preff`, we need to do the following:

1. Define the base algebra of the scoped effect, just like in the previous section.
2. Define the scoped algebra of the effect.
3. Embed the operations for use in **`PrEff`**.
4. Define a handler for the scoped-parameterised effect.

We show how to implement a scoped-parameterised effect on `processCustomers`. First, we introduce scoped effects without type parameters. Second, the scoped effect is extended to a scoped-parameterised effect, improving type safety and demonstrating the capabilities of `preff`.

5.2.1 Scoped Effects and Handlers in `preff`

We introduce the effect **`CustomerStore`** which is conceptually identical to **`CustomerDb`**, but is extended by a new operation. This new operation performs an existence check for a database before the procedure reads from it, and continues only if the database location can be found. In Haskell, such operations often follow the naming scheme `with`$\star$ where $\star$ refers to the name of the operation. Thus, we define the operation **`WithStore`**. Further, the effect **`CustomerStore`** defines the algebraic operations **`ReadStore`** and **`WriteStore`**, with syntax and semantics equivalent to the operations **`ReadCustomers`** and **`WriteCustomers`** from **`CustomerDb`**.

First, we define `processCustomersImproved` in Listing 5.9. It performs the same operations as `processCustomers`, but validates that the database `input` exists before it reads the customer data. The major changes lie in the type signature of `processCustomersImproved`:

- There is only a single **`Member`** constraint, for the **`CustomerService`** effect, since it is the only algebraic effect in `processCustomersImproved`.
- The type parameter `s` of **`PrEff`** is replaced by **`CustomerStore`**. It is the scoped effect for this procedure.

```
processCustomersImproved ::
  Member CustomerService f =>
  FilePath ->
  FilePath ->
  PrEff f CustomerStore () () ()
processCustomersImproved input output = do
  withStore input $ do
    customers <- readStore input
    newCustomers <- process customers
    writeStore output newCustomers
```

Listing 5.9: Example: `processCustomersImproved` using the scoped-parameterised effect **`CustomerStore`** to validate the existence of the `input` database location.

- Pre- and postcondition have been changed to `()`. The exact semantics of the pre- and postcondition are dependent on the scoped-parameterised effect, in this case **`CustomerStore`**. We choose the following semantics: This procedure can always be invoked and doesn't change the precondition during execution. The pre- and postcondition indicate that the procedure can only be invoked if the precondition is `()` and after this procedure is invoked, the postcondition is `()`. This allows us to invoke `processCustomersImproved` twice in a row, for example.

The body of `processCustomersImproved` has been changed as well, we perform an existence check via `withStore` which only continues execution if `input` exists. Otherwise, the implementation remains conceptually unchanged. With the existence check, `processCustomersImproved` is always safe to invoke and requires no further considerations from the developer.

To define the **`CustomerStore`** effect, we have to define a *scoped* effect. Scoped effects consist of two components, a base algebra and a scoped algebra as introduced by Piróg et al. (2018). In `preff`, scoped effects are different compared to other effect systems, since they are always parameterised by type-level proofs. These type-level proofs allow developers to improve type safety when desired, but there is boilerplate involved for defining scoped effects in simple cases. For example, `s`, `p` and `q` have to be given in the type signature of **`PrEff`** `f s p q a`, even when these are not used. The base algebra component for **`CustomerStore`** is similar to **`CustomerDb`**, as can be seen in Listing 5.10. The base algebra of a scoped effect requires at least three type parameters, `p` is the precondition of this effect, `q` the postcondition and `a` is the type of the value an operation produces. For now, both operations of **`CustomerStore`** do not impose any type-level restrictions, and thus we choose `()` for the pre- and postcondition.

To define the scoped algebra, the effect needs to have an instance of the **`ScopeE`** type family. This type family can be quite intimidating due to its seeming complexity. However,

```
data CustomerStore p q a where
  ReadStore :: FilePath -> CustomerStore () () [Customer]
  WriteStore :: FilePath -> [Customer] -> CustomerStore () () ()
```

Listing 5.10: Example: Base algebra for the scoped effect **CustomerStore** without type-level information.

this complexity is often not needed, and only a subset of its features is required in our case. Details of **ScopeE** and scoped-parameterised effects have been discussed in Section 4.2. In Listing 5.11 we show how to define the scoped algebra for **CustomerStore**. The type variables of **ScopeE CustomerStore** `m p q' q' q x' x` are defined as follows:

- **m**: Captures the monadic context of **PrEff** `f s`. In fact, `m` *is always* **PrEff** `f s`, but we cannot encode this directly since the effect signature `f` is not known at this point in execution. Thus, we assume some generic parameterised monad `m` for effect definition. Note, `m` must be a parameterised monad because it has three type parameters, a precondition, postcondition and the result type. Practically, this is a functional procedure that the effect **CustomerStore** scopes over. Relating to Listing 5.9, the monadic computation `m () () ()` is the second parameter to `withStore`/**WithStore**. The pre- and postcondition of `m` are both `()`.
- **p**, **q**: These are the pre- and postcondition of the program, where `p` needs to hold for the scoped operation to be permitted and `q` holds after execution. We have defined that **WithStore** can always be invoked and, after execution, the postcondition is the same as the precondition. There is no reason to require the pre- and postcondition to be `()`, as it is overly restrictive for no benefit. Thus, the type signature uses `p`, i.e. **ScopeE CustomerStore** `m` **p** `() ()` **p** `() ()`.
- **p'**, **q'**: These are the pre- and postcondition of the scoped region. In this case `m p' q' ()`. Intuitively, the scoped operation sets up the environment such that `p'` holds in the scoped region, and we require `q'` to hold after the inner operation is executed. Since **CustomerStore** requires no special pre- or postcondition, we choose `()` for both `p'` and `q'`. This is reflected in the final output type of **WithStore**, e.g. **ScopeE CustomerStore** `m p` **()** **()** `q () ()`.
- At last, **x'** and **x** are the result types of the respective operations. The former is the result type of the scoped operation, e.g. `m () () ()`, thus, `x' ~ ()`, while the latter is the final result type of `withStore`/**WithStore**. Since neither the inner nor outer computation produces a result, we specify `x ~ ()`. Thus, we obtain the type signature **ScopeE CustomerStore** `m p () () q` **()** **()**.

Identically to algebraic effect operations, the operations of a scoped effect need to be embedded into **PrEff**. The implementation, as can be seen in Listing 5.12, is similar

```haskell
data instance ScopeE CustomerStore m p p' q' q x' x where
  WithStore ::
    FilePath ->
    m () () () ->
    ScopeE CustomerStore m p () () p () ()
```

Listing 5.11: Example: Scoped algebra for the scoped effect `CustomerStore` without type-level requirements.

```haskell
readStore ::
  FilePath -> PrEff eff CustomerStore () () [Customer]
readStore p =
  sendP (ReadStore p)

writeStore ::
  FilePath -> [Customer] -> PrEff eff CustomerStore () () ()
writeStore p c =
  sendP (WriteStore p c)

withStore ::
  FilePath ->
  PrEff eff CustomerStore () () () ->
  PrEff eff CustomerStore p p ()
withStore i m =
  sendScoped (WithStore i m)
```

Listing 5.12: Example: Embed the operations of a scoped algebra for the effect `CustomerStore` as a monadic `PrEff` computation.

to algebraic effects but it requires two new helper functions: `sendP` and `sendScoped`. Conceptually, they serve the same purpose as `send` from Listing 5.2, but are specific to embedding operations of the respective scoped effect, as these affect the pre- and postcondition. Embedding operations for a scoped-parameterised effect has the same requirement as for algebraic effects, namely that the type signature of the function needs to correspond closely to the operation definition. For example, the result type of `readStore` has to be the same as the result type of `ReadStore`.

At last, a handler for `CustomerStore` needs to be defined. Unfortunately, the previously defined utility functions such as `interpret` and `interpretStateful` cannot be reused, since `CustomerStore` consists of two components, the base algebra and the scoped algebra. In `preff`, we define utility functions that abstract over `PrEff`, enabling developers to write handlers more easily and independently of the internal implementation

```haskell
runCustomerStoreIO ::
  Member (Embed IO) f =>
  PrEff f CustomerStore p q a ->
  PrEff f IVoid () () a
runCustomerStoreIO =
  interpretScopedH
    do \case
        ReadStore fp -> do
          embed $ readCustomersIO fp
        WriteStore fp cs -> do
          embed $ writeCustomersIO fp cs

    do \run -> \case
        WithStore fp m -> do
          exists <- embed (customersExistIO fp)
          when exists $ do
            run m
```

Listing 5.13: Example: Handler for the effect **CustomerStore** using the handler utility function **interpretScopedH**. The function argument `run` is used to conditionally run the scoped region if the database location is found.

of **PrEff**. Listing 5.13 shows the handler for the scoped effect **CustomerStore**. The implementation uses **interpretScopedH**, one of the many utility functions in `preff`, for handling scoped-parameterised effects. It splits the implementation into base and scoped algebra handlers, and resembles `interpret` but for scoped effects. While the handler for the base algebra is quite familiar by now, e.g. for each operation the semantics are defined using appropriate **IO** procedures, the scoped operation has the additional parameter `run`. This parameter allows the handler to decide how to continue program execution. Essentially, it is a reference to the effect handler we are defining, enabling us to re-use the semantics of the base and scoped algebra for the monadic region. In this particular example, it allows us to encode that the continuation of `withStore` is only executed if `customersExistIO` can find the store location.

The definition and implementation of the scoped effect **CustomerStore** is complete with the definition of the handler. excludes run-time failures, and provides a safe to use interface. However, the existence check is performed every time the procedure `processCustomersImproved` is invoked. This might be undesirable due to performance considerations, and moreover, does not make the usage of `readStore` and `writeStore` itself any safer. Developers may just use the operations without calling `withStore` for the input parameter. Scoped-parameterised effects can enrich the effect to make `readStore` always safe to use.

5.2.2 Scoped-parameterised Effects and Handlers in `preff`

Our algebraic effect system library `preff` provides pre- and postconditions in its effect monad `PrEff` which differentiates it from most other algebraic effect system libraries in Haskell. This enables developers to encode non-trivial proofs at the type-level, which are verified by the compiler at compile-time. In `preff`, we name these proofs *protocol types* as they can encode protocols like contracts for control flow execution.

Previously, we introduced scoped effects for `preff` on the `CustomerStore` example to implement the functional procedure `processCustomersImproved`. While this avoids run-time failures by introducing an existence check for the database location, it is still possible to use `readStore` incorrectly. In this section, we amend the scoped effect `CustomerStore` to make use of the pre- and postcondition sections provided by `PrEff` to increase the type safety and exclude incorrect usage.

Since it should only be possible to invoke the operation `ReadStore` when there is a proof for the existence of the database, we need to change the algebraic effect definition of `CustomerStore`. We introduce the phantom type `Store db`, which represents a proof that the database `db` exists. Thus, `ReadStore` is modified to expect a precondition `Store inp` and cannot be invoked if this precondition is not met. After reading the database, it remains valid, which means that after invoking the operation `ReadStore`, the proof still exists, i.e. the postcondition remains unaltered. Differently, `WriteStore` has no precondition, it can be invoked at any time. However, after writing to a certain location, we can be sure that this location exists. Thus, while `WriteStore` has no precondition, the postcondition is `Store out`, where `out` is the database location. Note that it is also possible to add the new location to a list of known database locations in the type signature. However, for the sake of simplicity we abstain from further advanced techniques. Eisenberg and Weirich (2012) provide an introduction to dependently typed programming in Haskell which can be translated for `preff`. We show the full definition of the base algebra in Listing 5.14.

Since `FilePath` is not sufficient to track the exact database location at the type-level, the database location parameter is replaced by `Proxy db`. `Proxy`[2] is often used in Haskell as a surrogate for communicating type information. The purpose is to use a type representation of the database location, for example `Proxy @"customers.db"`.

The extended `CustomerStore` effect suffices to implement `processCustomersTyped` which is the final version of `processCustomers` with improved type safety in `preff`. We show the implementation of `processCustomersTyped` in Listing 5.15. Interestingly, the implementation is now very similar to the original `processCustomers` from Listing 5.6. In both procedures the implementation assumes that the input database exists but previously the programmer had to verify correctness while now the compiler verifies type safety. The main difference is the type signature of `processCustomersTyped`:

[2] https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-Proxy.html

```haskell
data Store db

data CustomerStore p q a where
  ReadStore ::
    Proxy db -> CustomerStore (Store db) (Store db) [Customer]
  WriteStore ::
    Proxy db -> [Customer] -> CustomerStore p (Store db) ()
```

Listing 5.14: Example: Base algebra for scoped-parameterised effect **CustomerStore**. Uses a phantom type **Store** to encode pre- and postconditions for the operations.

- **Proxy inp/out**: Type representation of the database locations. It replaces the **FilePath** parameters but serves the same purpose: The procedure reads from the database location `inp` and writes new customer data to `out`. Naturally, at run-time, `inp` and `out` encode a concrete database location.

- **Store inp/out**: The functional procedure `processCustomersTyped` can only be invoked, if there is a proof that a database exists at `inp`. Further, after execution, the postcondition guarantees that the database at location `out` exists. This guarantee holds since the procedure `processCustomersTyped` uses `writeStore` which creates a database at the location `out`.

The implementation itself is identical to previous iterations, the only modification is the usage of **Ix.do** instead of the well-known **do**. This is a feature of recent GHC versions, called **QualifiedDo**[3] which allows developers to use the **do** syntax for monad-like structures that cannot implement the **Monad** type class. In Section 4.1, we show the implementation of **IMonad**, which provides the implementation of **Ix.do**. We require **Ix.do** because `processCustomersTyped` changes its type parameters during execution, from **Store** `inp` to **Store** `out`. Such an interface cannot be implemented with **Monad** only, but requires the **IMonad** interface. As a rule of thumb, whenever the pre- or postcondition changes during execution, the code region requires **Ix.do** instead of **do**.

Currently, given the algebraic component of **CustomerStore**, the only way to create a proof that a database location exists is by invoking **WriteStore**. However, this entails that we can never read from a store location unless we have written to it before. This is dissatisfactory, rather it should be possible to *check* whether a database location exists by encoding this proof for a scoped region. Such proofs can be achieved via the scoped operation **WithStore**, as shown in Listing 5.16.

[3] `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/qualified_do.html#extension-QualifiedDo`

```haskell
processCustomersTyped ::
  Member CustomerService f =>
  Proxy inp ->
  Proxy out ->
  PrEff f CustomerStore (Store inp) (Store out) ()
processCustomersTyped inp out = Ix.do
  customers <- readStore inp
  newCustomers <- process customers
  writeStore out newCustomers
```

Listing 5.15: Example: Type-safe variant of `processCustomers` that uses the scoped-parameterised effect `CustomerStore` to safely access the customer database. It requires `Ix.do` as it changes the pre- and postcondition of `PrEff`.

```haskell
data instance ScopeE CustomerStore m p p' q' q x' x where
  WithStore ::
    Proxy inp ->
    m (Store inp) q' () ->
    ScopeE CustomerStore m p (Store inp) q' p () ()
```

Listing 5.16: Example: Scoped-parameterised algebra for `CustomerStore`. The highlighted inner monadic region assumes that `Store inp` exists and may read its data.

The definition of `WithStore` is almost identical, except for the pre- and postcondition of the monadic region. In `m (Store inp) (Store out) ()`, the monadic computation can assume that the database exists and use `readStore` as required, since the precondition `Store inp` is satisfied by `WithStore`. After the execution of `m`, any arbitrary postcondition may hold, specified by `q'`. The operation `WithStore` discards the postcondition of the inner computation, and restores the initial precondition.

With the scoped-parameterised effect `CustomerStore`, we now consider an example where we invoke `processCustomersTyped` in Listing 5.17. For brevity, we assume the embedding functions have already been defined, as they follow the same schema as before. The example illustrates, how we have moved the existence check out of `processCustomersTyped`. This simplifies the implementation of the procedure `processCustomersTyped` and excludes issues at run-time as no developer can invoke `processCustomersTyped` without providing a proof of the database's existence.

At last, we have to define the handler for the scoped-parameterised effect `CustomerStore`. Like before, in Listing 5.13, the utility function `interpretScopedH` is used for conveniently defining the semantics of the base and scoped algebra of `CustomerStore`. Only one change is required, since the database location is only encoded in the type, we

```haskell
invocationExample ::
  Member CustomerService f =>
  PrEff f CustomerStore p p ()
invocationExample = Ix.do
  withStore (Proxy @"customers.db") $ Ix.do
    processCustomersTyped (Proxy @"customers.db")
                          (Proxy @"newCustomers.db")
```

Listing 5.17: Example: Invoke `processCustomersTyped` in a type safe manner.

```haskell
runCustomerStoreIO ::
  Member (Embed IO) f =>
  PrEff f CustomerStore p q a ->
  PrEff f IVoid () () a
runCustomerStoreIO =
  interpretScopedH
    do \case
        ReadStore (p :: Proxy inp) -> do
          let fp = symbolVal p
          embed $ readCustomersIO fp
        WriteStore (p :: Proxy out) cs -> do
          let fp = symbolVal p
          embed $ writeCustomersIO fp cs

    do \run -> \case
        WithStore p m -> do
          let fp = symbolVal p
          exists <- embed (customersExistIO fp)
          when exists $ do
            run m
```

Listing 5.18: Example: Final handler for the scoped-parameterised effect **`CustomerStore`**.

need to translate it into a value, e.g. a **`FilePath`**. GHC provides `symbolVal`[4], which translates the type to a **`String`** value. Once a value is created, the internal semantics of the effect can be specified using the existing procedures.

By now, we have shown how to define new algebraic effects in `preff`, give their operations

[4]`https://hackage.haskell.org/package/base-4.18.0.0/docs/GHC-TypeLits.html#v:symbolVal`

semantics and how to use them with **`PrEff`**. Additionally, `preff` is capable of expressing scoped algebras, similar to comparable algebraic effect systems in Haskell. Handlers are defined using convenient utility functions that trivialise defining new effects. Moreover, we showcased the scoped-parameterised effects which are unique to `preff`. Scoped-parameterised effects allow improving the type safety of a program by encoding the pre- and postcondition in the type signature of the effect monad **`PrEff`**. Even though scoped-parameterised effects are more complex than simple algebraic effects, they are opt-in in the implementation. They enable developers to use the simpler algebraic effects when appropriate, but also to improve type safety by introducing pre- and postconditions into their functional procedures.

5.3 Show Case: Session Types in `preff`

As a show case of using `preff`, we provide a novel encoding of binary session types into a scoped-parameterised effect. This requires less than 120 lines of code and indicates the expressiveness of `preff`. Later on, in Section 6.1, we will show that the expressivity of our new implementation of session types is comparable to the ones of related implementations of sessions types in Haskell. We argue that our new binary session type encoding benefits from the composability and extensibility of `preff` and that its practical applicability surpasses those of related session type libraries in Haskell. The full code of 114 lines of the **`Session`** effect of our session type implementation is shown in the Appendix.

The purpose of the session type effect is to ensure the communication between exactly two participants is well-typed. If a program is well-typed, then it can never deadlock and no unexpected messages are sent. This property must be verified by the compiler at compile-time, not at run-time. For this case study, we define a scoped-parameterised effect that tracks the order of its operations. Then we provide a pure execution implementation using cooperative multitasking, i.e. Coroutines, verified to never deadlock.

Session types were first introduced by Takeuchi et al. (1994) and have since then been a topic of extensive study. They are based on the π-calculus by Milner et al. (1992) which is a calculus for process communication. In the calculus, any reducible expression shows well-formed communications. Consequentially, session types ensure that a well-typed program does not deadlock. This property can be verified by a compiler using a calculus reminiscent of the π-calculus.

For embedding session types into Haskell, we first define what operations are valid in our session type embedding. We provide first-class support for the following operations:

- *Send* and *Receive* for messages.
- *Choice* and *Offer*, where one participant offers two branches and the other participant chooses one of them.

```haskell
data S t;     data C a b;     data SL body;
data R t;     data O a b;     data CL body;

type family   Dual' proc
type instance Dual' (R a)   = S a
type instance Dual' (S a)   = R a
type instance Dual' (O a b) = C (Dual a) (Dual b)
type instance Dual' (C a b) = O (Dual a) (Dual b)
type instance Dual' (CLU a) = SLU (Dual a)
type instance Dual' (SLU a) = CLU (Dual a)

type family Dual proc where
  Dual (x: xs) = Dual' x : Dual xs
```

Listing 5.19: Session type operations represented as type constructors. The types represent the operations **S**end, **R**eceive, **C**hoice, **O**ffer, **S**erver-**L**oop and **C**lient-**L**oop. The duality of the session type is implemented via the type families `Dual'` and `Dual`.

- *Loop* for repeating parts of a protocol. Our embedding provides unbounded loops, where one of the participants decides after each iteration of the loop if it shall terminate.

We express these operations as Haskell phantom types, as shown in Listing 5.19. Since these are phantom types, none of the data types have any value constructors, only type constructors, which are important for specifying the protocol in the pre- and postcondition of `PrEff`.

`S` and `R` represent *send* and *receive* operations respectively, while `C` and `O` are the *choice* and *offer* operators. The types `SL` and `CL` are used for encoding repetition of a protocol, e.g. loops. There are two different constructors, one for the *server* and one for the *client* of the loop, where the former decides when the loop ends.

We define a *protocol* as a type-level list of the operations defined in Listing 5.19. For example, a simple ping-pong client, that first sends a `String` and then receives a `String` is represented as:

```haskell
'[S String, R String]
```

A protocol that offers a choice between receiving a `String` and receiving an `Int` is:

```haskell
'[O '[R String] '[R Int]]
```

```haskell
type Session ::
  forall k. [k] -> [k] -> Type -> Type
data Session p q r where
  Send :: a -> Session (S a : p) p ()
  Recv :: Session (R a : p) p a
```

Listing 5.20: Base algebra of the `Session` effect.

At last, a server implementing the well-known "guess a number" game generates a number at random, repeatedly waits for the client to guess a number until it guesses correctly is modelled as:

```haskell
'[SLU '[R Int]]
```

To make sure two participants are compliant, e.g. one protocol is the *dual* of the other, we define for each operation its dual. For example, when the client expects to receive a number, then the server needs to send a number. We implement an open type family, named `Dual'`, which defines for each operation its dual. Since a protocol is a list of operations, we define a closed type family `Dual` that applies `Dual'` to each operation of the protocol. We show the full implementation of the duality relation in Listing 5.19. Note, this definition of duality is not complete and may reject programs that are in fact well-formed. Other notions of duality are discussed by Bernardi et al. (2014), but we decide to use the inductive duality which is sufficient for our example.

After defining the operations and duality of the dual-party session type effect, we introduce the base algebra of the `Session` effect in Listing 5.20. The base algebra consists solely of the operations `Send` and `Recv`. In similar spirit to previous examples, the operations in the precondition indicate that they can only be executed, if the respective operation is expected to be executed next. The operation is then removed from the operation list, i.e. removed from the postcondition. Intuitively, a program starts from a protocol definition and incrementally consumes parts of the precondition until it has reached the end, i.e. the postcondition is `'[]`. If a protocol fully consumes the protocol, we say a program is compliant with the protocol.

The scoped algebra for `Session` implements five operations. While the implementation is shown in Listing 5.21, we discuss each constructor in detail.

- `Offer`: Provides a binary choice to the communication partner. Since the exact path in the protocol is only known at run-time, the operation needs to be able to handle both branches in the protocol. Hence, it takes two monadic parameters, which may have a different protocol, signalled by `a` and `b`. However, both protocols need to be fully consumed and thus the postcondition is `'[]` for both branches.

```
data instance ScopeE Session m p p' q' q x' x where
  Offer ::
    m a '[] x ->
    m b '[] x ->
    ScopeE Session m (O a b : c) '[O a b] '[] c x x
  Sel1 ::
    m a '[] x ->
    ScopeE Session m (C a b : c) a '[] c x x
  Sel2 ::
    m b '[] x ->
    ScopeE Session m (C a b : c) b '[] c x x
  LoopS ::
    m a '[] (Maybe x) ->
    ScopeE Session m (SLU a : c) a '[] c (Maybe x) [x]
  LoopC ::
    m a '[] x ->
    ScopeE Session m (CLU a : r) a '[] r x [x]
```

Listing 5.21: Scoped algebra of **Session**.

- **Sel1**/**Sel2**: When one of the communication partners offers a binary branching choice in the protocol, either of the branches can be chosen at run-time by the other communication partner. Hence there are two distinct operations for deciding on the first or second protocol branch respectively. Since each operation follows a single branch, only one monadic parameter compliant with the respective protocol branch is required for either operation. These two operations form the counterpart to the **Offer** operation.

- **LoopS**: Handles the server-side of the loop construct. It takes a monadic action that represents the body of the loop and the body is repeatedly executed producing a value of type **Maybe** x. If the loop body produces a **Nothing** value, the loop ends for both participants. Thus, the server controls how many times a loop is executed. All loop results are collected in a list, namely [x].

- **LoopC**: Handles the client-side of the loop construct and it takes a monadic action that represents the body of the loop. As the client-side can not terminate the loop on its own, each loop iteration may produce a value of type x. Similar to before, the results of each iteration are collected in a list [x].

We embed the operations as monadic **PrEff** computations identically to previous examples, such as Listing 4.11. Further, we provide example programs that are compliant with the session types presented above in Listing 5.22.

```haskell
-- Implements a simple ping-pong server
pingPong ::
  PrEff f Session '[S String, R String] '[] String
pingPong = Ix.do
  send "Ping"
  s <- recv
  pure s

-- Offer a choice between receiving a 'String' and an 'Int'.
stringOrInt ::
  PrEff f Session '[O '[R String] '[R Int]] '[] String
stringOrInt =
  offer
    Ix.do
      n <- recv @String
      pure n

    Ix.do
      n <- recv @Int
      pure (show n)

-- Game server where the client tries to guess a number.
-- Assumes 'RandomNumber' effect that has a single operation:
--
--   * 'getNumber': generates a random number.
--
-- The procedure 'guessNumberServer' returns the number of
-- attempts it took the client to guess the number.
guessNumberServer ::
  Member RandomNumber f =>
  PrEff f Session '[SLU '[R Int]] '[] Int
guessNumberServer = Ix.do
  num <- getNumber
  attempts <- loopS $ Ix.do
    n <- recv
    if n == num
      then pure Nothing
      else pure (Just ())
  pure $ length attempts
```

Listing 5.22: Example: Monadic procedures implementing various protocols in `preff`.

This ends the tour of `preff`. We have shown a complete case study of how to define and use effects using `preff`.

CHAPTER 6

Evaluation

In this section, we evaluate our algebraic effect system `preff` with respect to expressivity and performance. First, we evaluate the *expressiveness* of `preff` by comparing our lightweight session types encoding, as introduced in Section 5.3, to existing session type libraries in Haskell based on their respective features. Second, we examine our effect system library's real-world applicability, by evaluating the *performance* comparing it to well-known state-of-the-art effect system libraries.

6.1 Expressivity

Regarding expressivity, we qualitatively compare our implementation of session types with related libraries available in Haskell according to the following metrics.

1. *Control flow patterns*: Which control flow patterns are supported by the library? For example, is protocol alternation supported, and how can parts of a protocol be repeated? In particular, repetition of protocol parts can be implemented in various ways with different trade-offs.

2. *Communication layouts*: Which communication layouts are supported by the library? We distinguish between these three layouts:

    - *Two communication partners only*: The session type is capable of verifying the communication between exactly two actors can never deadlock. Moreover, there can only be one communication channel.
    - *More than two protocol participants*: The communication between more than two actors can never deadlock and the session type provides a global context that ensures this property over multiple communication channels. This is a strictly more expressive session type.

- *Multiple point-wise dual participants*: While there are multiple communication channels such that any two participants can safely communicate, there is no global context. Thus, deadlocks are possible in specific scenarios, such as the "Dining Philosopher's" problem proposed by Dijkstra. However, this is a strict superset of supporting exactly two communication partners.

3. *First-class citizen session type*: Is the session type a first-class citizen in the library? This is the case if it is possible to delegate the handling of a session type to a different actor. Often, this is referred to as channel delegation.

4. *Extensible implementation*: Can the library be extended with additional effectful computations? For example, a library is extensible if it provides a monad for writing a program adhering to a session type that can perform computations with side effects, such as `IO` computations. In general, an extensible session types library requires a monad transformer interface, or needs to be part of an effect system.

Due to various changes to the compiler internals of GHC, most of the related libraries do not compile any more with recent versions of GHC, such as `9.6.2`. The only libraries that can be currently compiled are `effect-sessions`, `simple-sessions`, `Coroutine` and our contribution `preff`. Migrating the remaining libraries to be compatible with the recent GHC version `9.6.2` is out of scope of this thesis. Thus, we take examples from the original papers and assume their correctness. This leaves a margin of error, but small inconsistencies do not distract of this work, and are therefore negligible.

6.1.1 Findings and Discussion

Table 6.1 summarises the control flow primitives supported by the various implementations of session types. They all session type libraries support some form of `send` and `receive`, and protocol alternations. The main difference lies in how repetition is implemented, either by using recursion or loops. Many session type calculi are based on the π-calculus, thus encodings tend to use recursion variables and fixed-points to simulate equi-recursive types (Dardha, 2014). Notable exceptions are `Coroutine` and `preff`, which both support *looping* constructs but not recursion. While recursion variables are a more faithful translation of the π-calculus, loops are more naturally expressed in `preff` and are, to the best of our knowledge, equally expressive as recursive encodings.

In Table 6.2, we summarise the possible communication styles and whether a session type is a first-class citizen in the respective library. No library provides session types capable of verifying a protocol between multiple communication partners. However, session types are first-class citizens in all libraries that support more than dual party session types.

At last, the underlying implementation of session types plays a vital role when it comes to flexibility and composability with other frameworks and libraries. We present an an overview of the implementations in Table 6.3. Further, we elaborate the different implementations and discuss their advantages and disadvantages. Note, most libraries

Table 6.1: Control flow patterns supported by session type libraries in Haskell.

Framework	Alternation	Recursion	Loops
`simple-sessions`	✓	✓	✗
`full-sessions`	✓	✓	✗
`sessions`	✓	✓	✗
`effect-sessions`	✓	✓	✗
`sessiontypes`	✓	✓	✗
`Coroutine`	✓	✗	✓
`preff`	✓	✗	✓

Table 6.2: Communication patterns and capabilities of session type libraries in Haskell.

Framework	Communication Partners	First-Class Citizen
`simple-sessions`	Multi point-wise dual	✓
`full-sessions`	Multi point-wise dual	✓
`sessions`	Multi point-wise dual	✓
`effect-sessions`	Multi point-wise dual	✓
`sessiontypes`	Dual	✗
`Coroutine`	Dual	✗
`preff`	Dual	✗

define a type **`Session`** for embedding session types into a monadic interface. These definitions usually differ in each respective library. Unless stated otherwise, when we refer to the type **`Session`**, we refer to the definition in the library we are currently discussing.

`simple-sessions` Introduced by Pucella and Tov (2008), `simple-sessions` uses a parameterised monad wrapping the **`IO`** monad. Thus, it extends the **`IO`** monad with two phantom parameters to implement the interface of the parameterised monad. Similar to the already mentioned library `Coroutine`, there is no way to extend the **`Session`** monad of `simple-sessions`. It is, therefore, not possible to add effects to the **`Session`** monad for which the authors have not already accounted for.

`full-sessions` The `full-sessions` was implemented by Imai et al. (2010). Its **`Session`** monad wraps a **`State`** monad on top of the **`IO`** monad. While **`Session`** implements the pendant of a parameterised monad, contrary to other session type libraries, pre- and postconditions do have a run-time representation. They represent the current state of various session types, implementing a state machine for advancing the protocol. Due to the run-time representation, session types can be easily delegated to other actors. However, since **Session** is neither a monad transformer nor provides any other means of extension, there is no way to extend it with effects the authors did not anticipate before. Adding additional effects is impossible in the current implementation, only **IO** effects are permitted.

sessions One of the oldest session type embeddings in Haskell, introduced by Neubauer and Thiemann (2004). Its **SessionType** type is essentially a **State** monad, carrying a value representation of the current session. Since there are no points of extension or transformer interfaces, it is not possible to extend `sessions` to embed other monadic computations other than **IO**.

sessiontypes A different implementation is given in `sessiontypes`[1]. In addition to leveraging parameterised monads, `sessiontypes` features the additional type class **MonadSession**. This enables different interpretations of a session type, in both pure and impure code. The implementation uses value terms for embedding session type operations, building an abstract syntax tree for the communication process. By choosing an algebraic data type encoding for session type operations, it bears close resemblance to a free monad encoding. It also provides the type class **IxMonadT**, to turn instances of **MonadSession** into an indexed monad transformer. This enables developers to embed arbitrary monadic computations, such as **IO** and **State**, into the execution context of **MonadSession**. However, only one parameterised effect can be used, a limitation `sessiontypes` shares with `preff`.

effect-sessions The library `effect-sessions` was implemented by Orchard and Yoshida (2016) and uses a graded monad interface for its sequential computation interface. Its **Process** type wraps the **IO** monad and implements the graded monad interface.
Session types are encoded into the phantom parameter of **Process** and communication is performed via explicitly passed channel values. There are no extension points to **Process** type, thus the only other monadic operation that can be performed is the **IO** effect. While being a very sophisticated library that elegantly supports session type delegation and multiple point-wise communication participants, it proves to be incompatible with other existing frameworks, making it difficult to reuse existing code.

Coroutine In `Coroutine`[2] the authors use a parameterised continuation monad for its implementation. This permits a natural implementation of session types using a pure continuation monad. The library permits wrapping of arbitrary monadic operations into the context of the parameterised monad interface but introduces no way of extension. In other words, neither monad transformers, nor **IO** actions are permitted in combination with the session type monad. Although the encoding of session types is simple and insightful, its limitation renders it inflexible in practice.

[1] `https://hackage.haskell.org/package/sessiontypes`
[2] `https://hackage.haskell.org/package/Coroutine`

preff At last, preff encodes session types into an algebraic effect system. Since preff is designed for adding extensions, it is no surprise that the freer monad encoding of effects seems to be the most flexible one of the most commonly known session type encodings. It allows for embedding arbitrary effects, as known from other effect systems, into the communication between two participants, making it the most flexible solution for session types out of the considered competitors. Additionally, preff is extensible, it is possible to implement a custom embedding of session types, with support for currently unsupported control flow patterns, such as recursion, or even multiple communication participants.

Table 6.3: Underlying implementations of session type libraries in Haskell

Framework	Implementation
simple-sessions	Parameterised **IO** Monad
full-sessions	Parameterised **IO**/**State** Monad
sessions	Parameterised **State** Monad
sessiontypes	Parameterised Monad Transformer with Term Representation
effect-sessions	Graded **IO** Monad
Coroutine	Parameterised Continuation Monad
preff	Parameterised Freer Monad / **PrEff**

6.1.2 Conclusion

The session type encoding we chose to illustrate the expressiveness of preff is a flexible and lightweight alternative to most other session type libraries. It is designed for extensibility, e.g. allows adding more communication primitives, includes no run-time overhead, and can perform other effects beyond **IO** computations. Additionally, the encoding has better type inference, due to lower complexity and requires no experimental features in GHC. The sole drawback is that this encoding permits no obvious session type delegation. However, we think this drawback is not inherent to preff's design and could be improved in subsequent work.

Based on this qualitative comparison, we conclude preff excels at being an algebraic effect system that provides extraordinary flexibility to implement complex control flow patterns, such as dual-party session types.

6.2 Performance

Run-time performance is important to algebraic effect system libraries as they are intended to provide:

- a high-level description of what a program does, and

- primitives that allow implementing performant programs and algorithms.

However, all algebraic effect system libraries incur performance overhead due to their effect monad implementation and effect handler fusion. We measure the performance of well-known effect system libraries, including our contribution `preff`, on a series of microbenchmarks.

6.2.1 Methodology

The metric of *performance* is measured based on the `effect-zoo`[3] project. It implements microbenchmarks for commonly used effect system libraries in Haskell. We add `preff` to the set of benchmarked effect system libraries in our fork [4] `effect-zoo` and report our findings. First, we introduce the microbenchmarks and discuss what metric they are supposed to benchmark. Each library implements the microbenchmark, but for brevity we only show the benchmark code for `preff`. The type signatures of `preff` microbenchmarks are specialised to indicate no parameterised effects are permitted using **`IVoid`**. Second, we present the benchmark results and discuss potential bottlenecks in `preff`'s implementation. Finally, we summarise the results and discuss whether `preff`'s performance is sufficient for real-world applications.

The benchmarks are executed using the benchmark framework `criterion`, which is fine-tuned for Haskell microbenchmarks. We run the experiments on an AMD Ryzen 7 3800X processor, which has a single core performance of 3.9 GHz up to 4.5 GHz, on a machine with 32 GB DDR4 RAM. We use GHC version `9.6.2` to compile the project and apply the compilation flags `-O2 -flate-specialise`.

CountDown

In our microbenchmark `CountDown`, a program counts down from an initial value until the value is zero. The implementation uses a single **`State`** effect with a mutable **`Int`** variable. `CountDown` measures the minimal overhead induced by the encoding of the effect system library, since no other effects can affect the overall performance. Ideally, running such simple programs is as quick as a pure function in Haskell. The code for `CountDown` is presented in Listing 6.4.

BigStack

The `BigStack` microbenchmark consists of one simple **`Reader`** and one **`State`** effect. The value of the **`State`** effect is repeatedly modified for a pre-determined number of times. In Listing 6.5, the program is shown as specifically written for `preff`.

The program is executed using a variable number of unused in-between handlers that handle effects that are not actually used by the program. It aims to benchmark how

[3] `https://github.com/ocharles/effect-zoo`
[4] `https://github.com/fendor/effect-zoo`

```
countDown ::
  Member (State Int) eff =>
  PrEff eff IVoid () () Int
countDown = do
  n <- get
  if n <= 0
    then pure n
    else do
      put (n - 1)
      countDown
```

Listing 6.4: Source code of the microbenchmark `CountDown` in `preff`.

```
bigStack ::
  (Member (State Int) f, Member (Reader Int) f) =>
  PrEff f IVoid () () ()
bigStack = Ix.do
  n <- ask
  replicateM_ n (modify (+ n))
```

Listing 6.5: Source code of the microbenchmark `BigStack` in `preff`.

unused effects impact execution performance. As the unused effects, the **`Identity`** effect is chosen. Additionally, the cost of traversing the abstract syntax tree in effect systems using the free monad encoding becomes apparent, since each handler has to traverse the full program structure. The benchmark is run with 0, 1, 5, and 10 **`Identity`** effects. We expect effect handler fusion to have a noticeable impact on this microbenchmark because it reduces the number of program traversals.

FileSizes

Effect systems are extensible by design: Adding new effects is the norm, not the exception. We measure the overhead of custom effects in the microbenchmark `FileSizes`. Some effect systems provide additional performance improvements for commonly used effects, such as **`State`** or **`Reader`**. Custom effects are not specifically optimised, thus, this measures how expensive interpreting a custom effect is on its own. This benchmark features **`Logging`** and **`File`** effects, the former writes messages to an **`IORef`** [5], while the latter reads the size of files from disk. Out of all our benchmarks, it has the

[5] `https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-IORef.html#t:IORef`

```haskell
fileSizes ::
   (Member File f, Member Logging f) =>
   [FilePath] ->
  PrEff f IVoid () () Int
fileSizes files = do
  sizes <- mapM calculateFileSize files
  return (sum sizes)

calculateFileSize ::
   (Member File f, Member Logging f) =>
  FilePath -> PrEff f IVoid () () Int
calculateFileSize path = do
  logMsg ("Calculating the size of " ++ path)
  msize <- tryFileSize path
  case msize of
    Nothing   ->
      0 <$ logMsg ("Could not calculate the size of " ++ path)
    Just size ->
      size <$ logMsg (path ++ " is " ++ show size ++ " bytes")
```

Listing 6.6: Source code of the microbenchmark `FileSize` in `preff`.

closest resemblances to a real-world application. Listing 6.6 contains the code of this microbenchmark.

Reinterpretation

A major difference of algebraic effect systems compared to the monad transformer based main competitor `mtl` is that they can "re-interpret" effects in terms of other existing effects. This allows to reduce code duplication since interpretation of an effect can be delegated to a more specialised effect, e.g. **`State`**, while still having the expressivity of a custom effect. Additionally, it allows developers to simulate the interpretation of effects for testing and addressing cross-cutting concerns, such as authentication, for deployment. The microbenchmark `Reinterpretation` features a **`Zooit`** effect which represents an abstract service for listing data. It reinterprets the effect into a mocked **`HTTP`** and **`Logging`** effect, which captures context information and error handling, without leaking implementation details to the user of **`Zooit`**.

6.2.2 Discussion of Findings

We present and discuss the results of the microbenchmark project `effect-zoo`.

Table 6.7: Execution time of `CountDown` in µs (lower is better).

	Size 100		Size 1000	
Name	Mean	σ	Mean	σ
`mtl (lazy)`	12.409	0.130	123.144	0.614
`mtl (strict)`	11.169	0.048	110.797	0.247
`freer-simple`	4.713	0.076	45.978	0.559
`fused-effects`	20.131	0.067	199.633	0.804
`polysemy`	22.992	0.124	228.982	1.468
`preff`	5.383	0.022	53.448	0.362
Reference	0.039	0.001	0.241	0.002

CountDown

The run-time figures are shown in Table 6.7 and a corresponding visualisation thereof in Figure 6.8. As expected, the *Reference* implementation, which uses no effect system or other abstractions, outperforms by a considerable margin. GHC can generate very efficient byte-code for this particular instance. As a surprise to us, the two second-fastest frameworks are `preff` and `freer-simple`, with almost identical performance, surpassing `mtl`, which we expected to be the fastest, since GHC is optimised to compile type classes into efficient code. The two entries for `mtl` are using a *lazy* and a *strict* implementation of the *State* monad respectively. In microbenchmarks, laziness is almost always a hindrance if every value is guaranteed to be needed. Delaying a computation adds an indirection, justifying the performance difference in the lazy and strict `mtl` implementations. We include both evaluation strategies in the evaluation. The slowest frameworks are `polysemy` and `fused-effects`, which both build on the theoretical foundation of Wu et al. (2014) and Wu and Schrijvers (2015). We conjecture the performance deteriorates because of the additional complexity introduced by scoped algebras. Handling scoped algebras introduces additional work for the algebraic effect system libraries `polysemy` and `fused-effects`, while `freer-simple` does not support scoped algebras. Even though `preff` supports scoped algebras, due to the separation of scoped effects and non-scoped effects, `preff` does not have to perform any additional work in this microbenchmark.

BigStack

The `BigStack` microbenchmark measures the performance impact of a larger effect list. Its results are presented in Table 6.9 and Figure 6.10. There is no *Reference* implementation this time because no such implementation exists. In this particular benchmark, `mtl` performs similarly to the algebraic effect system library `freer-simple`. We expected the fusion optimisations of `fused-effect` to have a bigger impact, but it performs similarly to `polysemy`. It is also surprising that `freer-simple` is the second-fastest effect system library. Discrepancy in expectation might be caused by high constant factors of the optimisations applied in `polysemy` and `fused-effect`. Underwhelming

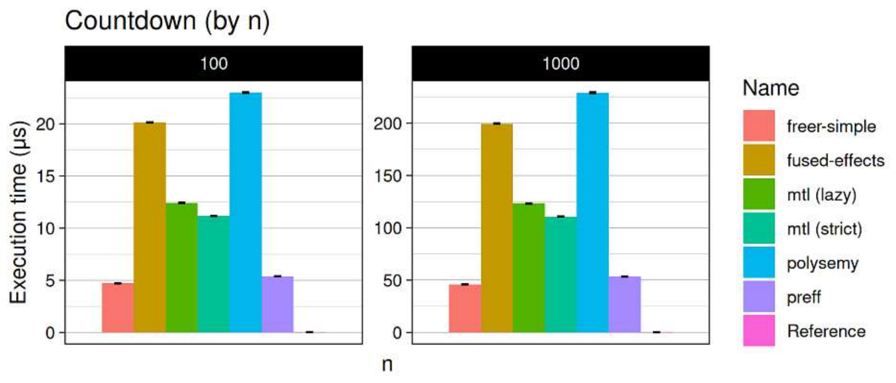

Figure 6.8: Visualisation of `CountDown` benchmark results.

are the results of `preff`: up to 32 times slower than `freer-simple`, and 13 times slower than `polysemy` and `fused-effect`. The results seem very suspicious because even in the case of zero unused **`Identity`** effects, the performance suffers greatly, while in Section 6.2.2 we see the performance of interpreting a single effect is competitive with other effect system libraries. Even more suspicious, the relative performance difference between `preff` with the stack sizes *0* and *10* is about 5 %, which is noticeable smaller compared to other effect system libraries. We identify two possible reasons:

1. Interpreting two effects incurs considerable performance overhead in `preff`.

2. Performance sensitive optimisations have not been applied by GHC.

Results in later sections render possibility 1 unlikely. Thus, we conjecture that GHC fails to optimise `preff` in this microbenchmark. Careful addition of **`INLINE`** pragmas and strictness annotations might mitigate the performance regression.

Further, we repeated the experiment for the GHC version `9.4.7` to compare results and noticed that `mtl` gets considerably better optimised. When investigating one of the intermediate representations of GHC, called `core`, it becomes obvious that this particular GHC version optimises monad transformers more efficiently. This supports our hypothesis of optimisation brittleness in microbenchmarks.

FileSizes

In Table 6.11, the results of the `FileSizes` benchmark are presented. Further, in Figure 6.12 a visual contextualisation is shown. The monad transformer based library `mtl` achieves identical performance to the *Reference* implementation. However, `fused-effect` also performs identically, making it the fastest effect system library in this benchmark. The other three libraries have similar performance characteristics, in the smaller instances undistinguishable, while in the largest instance, `freer-simple`

Table 6.9: Execution time of `BigStack` in µs (lower is better).

	Size 0		Size 1	
Name	Mean	σ	Mean	σ
`mtl`	163.118	1.074	169.193	1.098
`freer-simple`	122.967	0.688	144.100	0.958
`fused-effects`	370.047	1.096	387.629	1.451
`polysemy`	295.446	1.141	320.466	1.841
`preff`	3908.220	18.162	3925.283	25.996
	Size 5		Size 10	
Name	Mean	σ	Mean	σ
`mtl`	187.481	2.032	212.146	2.426
`freer-simple`	203.818	0.920	304.587	1.144
`fused-effects`	450.658	3.617	523.783	4.600
`polysemy`	453.051	2.453	609.322	3.121
`preff`	3962.599	17.959	4087.535	40.503

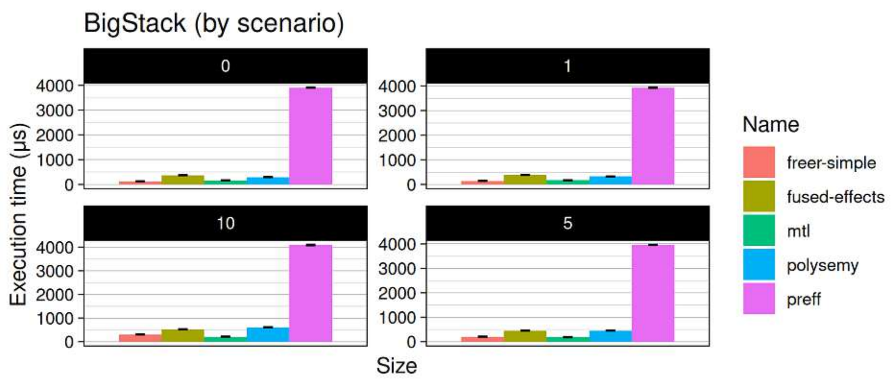

Figure 6.10: Visualisation of `BigStack` benchmark results.

pulls slightly ahead. While `preff` is not as fast as some of the competitors, it is not the slowest either, hinting there is more potential to further improve the performance. The relative closeness of results might be a hint, that this benchmark is bottlenecked by the number of disk operations, rather than the effect system itself. We think, this is a common situation for most real-world applications.

Reinterpretation

In the last microbenchmark `Reinterpretation`, the capability of reinterpreting effects in terms of other effects is tested for its usability in practice. Its performance measurements are shown in Table 6.13 and visualised in Figure 6.14. Note, neither *mtl* nor *Reference* are present here because the concept of reinterpretation is not expressible in either. Interestingly, performance is more or less the same for all effect systems, except for

Table 6.11: Execution time of `FileSizes` in µs (lower is better).

	Size 1		Size 10		Size 100	
Name	Mean	σ	Mean	σ	Mean	σ
`freer-simple`	2.099	0.007	20.196	0.175	202.616	1.668
`fused-effects`	2.886	0.013	26.332	0.067	261.763	0.972
`mtl`	2.257	0.007	20.802	0.109	206.104	0.601
`polysemy`	2.823	0.040	26.683	0.120	264.883	3.751
`preff`	1.918	0.020	20.070	0.250	259.661	1.389
Reference	1.591	0.012	15.504	0.027	155.088	0.634

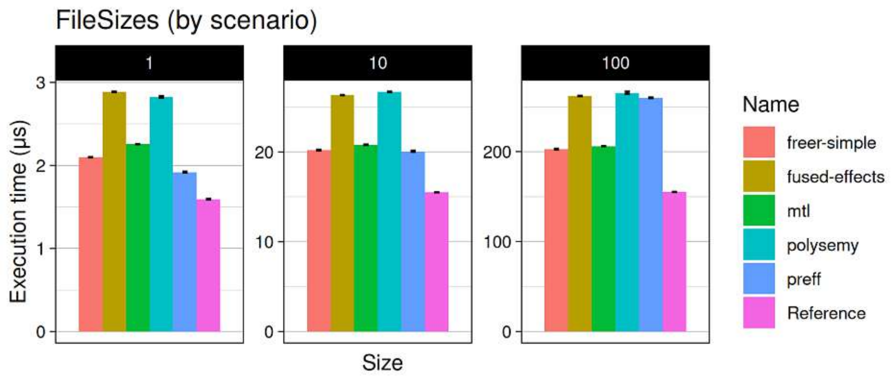

Figure 6.12: Visualisation of `FileSizes` benchmark results.

	Size 1		Size 10		Size 100	
Name	Mean	σ	Mean	σ	Mean	σ
`freer-simple`	1.334	0.008	13.698	0.048	133.648	0.307
`fused-effects`	2.018	0.008	20.483	0.056	217.852	0.269
`polysemy`	3.314	0.019	30.915	0.110	317.399	1.198
`preff`	1.344	0.013	13.707	0.091	165.651	0.377

Table 6.13: Execution time of `Reinterpretation` in µs (lower is better)

`polysemy`. It seems, `polysemy` has higher overhead in all instances, potentially from performing more work in the background.

6.2.3 Summary of Findings

All of the effect system libraries in this benchmark are used in software applications at the time of this writing. Since `preff`'s performance is competitive with the presented algebraic effect system libraries, we claim its adequacy for all software applications that are already using algebraic effect system libraries. The concern about optimisation brittleness remains for programs compiled with GHC. A further inspection of the microbenchmark `BigStack` might illuminate where `preff` currently falls behind in practice. Explicit

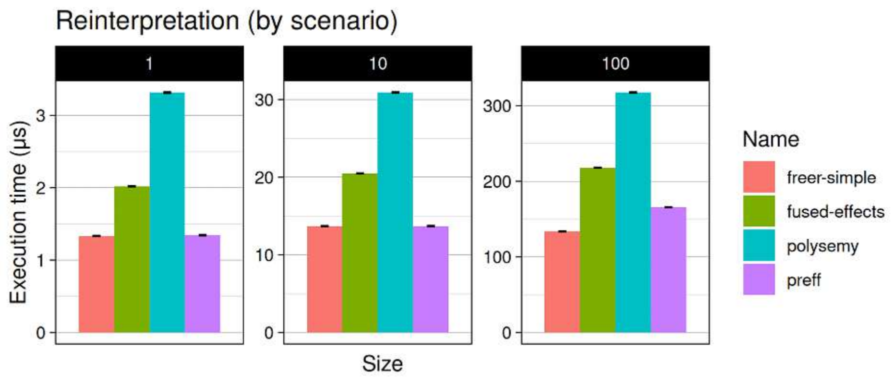

Figure 6.14: Visualisation of `Reinterpretation` benchmark results.

efforts to implement effect handler fusion similar to Wu and Schrijvers (2015) and improving the asymptotic complexity of **`PrEff`**'s monadic bind operator as shown by Voigtländer (2008) could narrow the gap in performance.

However, microbenchmarks are unsuited to estimate the performance in an application in general. They might benchmark a use-case the given effect system is particularly well-fitted to handle, or a pathological example that does not even arise in real-world applications. As such, we cannot extrapolate the performance of `preff` to specific real-world scenarios, and only tell what kind of scenarios are cheap and what are expensive.

We argue that `preff` is acceptable for real-world applications. Whereas it does not outperform competitors and falls behind in some aspects, its unique design trade-offs and future optimisation opportunities render it a promising alternative to current effect system libraries. Especially, since we believe the performance of most applications is mainly dependent on external sources, such as network bandwidth and disk read/write performance.

CHAPTER 7

Conclusion

In this thesis we introduced a new algebraic effect system library called `preff`. This library differs from traditional algebraic effect system libraries as it combines the strengths of traditional effect systems, such as composability, ease of use, and multiple interpretations per effect, with parameterised effects which significantly improve type safety of procedures. Additionally, we focused on usability of `preff`, providing various utilities to keep the non-parameterised effects easy to implement, while enabling developers to use parameterised effects in practice. Our qualitative and experimental evaluation, when compared to related state-of-the-art libraries in Haskell, proved `preff` to be an exceptionally expressive effect system library which performs well in practice.

Our new effect monad **`PrEff`** is the amalgamation of state-of-the-art effect monads. It takes the freer monad encoding known in traditional algebraic effect system libraries and extends it for scoped-parameterised effects, leading to the parameterised freer monad encoding. This enabled us to benefit from the related research on algebraic effect system libraries and infused them with additional type safety. Futhermore, we extended the `weave` abstraction for parameterised effects, allowing them to weave context through the monadic regions of scoped effects. The result of this is an algebraic effect system library that is as easy to use as related effect system libraries but can benefit from additional pre- and postconditions in the program execution. We demonstrated the expressiveness of our contributed library by implementing a novel encoding of session types that can be used in our algebraic effect system. Implementing session types required approximately 120 lines of code, which demonstrates that `preff` is well-suited for implementing complex and novel control flow patterns. Finally, we proved that `preff`'s run-time performance is competitive with related effect system libraries that are actively being used in modern software applications.

Conceptually, we extended the map of algebraic effect system libraries by introducing a combination of algebraic and scoped-parameterised algebraic effects and effect handlers, which, to the best of our knowledge, is not present in previous systems. This enables the implementation to provide additional type safety in procedures that previously had to be verified manually by the developer.

Our algebraic effect system library is published to the wider Haskell community by uploading `preff` to Hackage[1], where it can easily be shared, explored and experimented with. Additionally, to encourage future contributions from third-parties, we publish the source-code of `preff` on GitHub[2] under the BSD-3-Clause License.

7.1 Future Work

Our algebraic effect system library `preff` can still be improved in various directions in future research. We highlight below the most important ones.

Ecosystem

Writing an application requires access to the ecosystem of Haskell. Thus, `preff` needs to be compatible with well-known packages available on Hackage[3], the Haskell package repository. In particular, `preff` should provide support for common tasks such as logging, network requests and database access. Other effect system libraries provide support for common tasks in separate packages. Extending `preff` to provide similar support would help adoption in real-world applications.

Performance

As we have shown in Section 6.2, the run-time performance can be further improved. First, we could apply the techniques of Voigtländer (2008) to improve the asymptotic complexity of the monadic bind operation `(>>=)`. This should require minimal internal changes and no changes to the external interface. Initial experiments show promising results, considerably improving the run-time performance of the microbenchmark `BigStack`. Second, writing effect handlers to enable efficient fusion as shown by Wu and Schrijvers (2015) should further improve the performance.

Integration into the framework of van den Berg and Schrijvers (2023)

A general purpose framework, called “Higher-Order Effects & Handlers Framework”, for higher-order effects and handlers is introduced by van den Berg and Schrijvers (2023). Their framework provides a categorical formalisation of algebraic effect systems with various feature sets and implements them generically. Integrating `preff` into the framework would allow us to benefit from the categorical foundations and generic performance improvements.

[1]`https://hackage.haskell.org/package/preff`
[2]`https://github.com/fendor/preff`
[3]`https://hackage.haskell.org/`

Dependent Types

In this thesis, we have shown advanced typing techniques to verify session types. Future work may explore the usage of dependent types in `preff` using singletons as introduced by Eisenberg and Weirich (2012).

Scoped Effects

Our algebraic effect system library `preff` is currently limited to a single scoped effect to be used for a procedure. This limitation is pragmatically motivated and ensures the type safety of the algebraic effect system. However, allowing the usage of more scoped effects when rigorous restrictions are in place is possible. For example, it might be feasible to allow additional scoped effects if the scoped-parameterised effect is provably not being used in a procedure and the control flow manipulation of the scoped effect can not escape the procedure context.

List of Figures

List of Tables

List of Listings

Acronyms

AST abstract syntax tree. 5, 34, 35

GADT generalised algebraic data type. 15, 23, 25, 32, 33, 38, 45, 46, 50

GHC Glasgow Haskell Compiler. 1, 5, 6, 21, 22, 59, 61, 70, 73, 74, 77, 78, 80

Bibliography

R. Atkey. Algebras for Parameterised Monads. In A. Kurz, M. Lenisa, and A. Tarlecki, editors, *Algebra and Coalgebra in Computer Science*, pages 3–17, Berlin, Heidelberg, 2009a. Springer Berlin Heidelberg. ISBN 978-3-642-03741-2. doi: 10.1007/978-3-642-03741-2_2. URL `https://link.springer.com/chapter/10.1007/978-3-642-03741-2_2`.

R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009b. doi: 10.1017/S095679680900728X. URL `https://www.cambridge.org/core/journals/journal-of-functional-programming/article/parameterised-notions-of-computation/82CE5F0583C3390BBBD305830255FAA0`.

F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1086. URL `https://www.sciencedirect.com/science/article/pii/S0890540185710863`.

A. Bauer and M. Pretnar. An Effect System for Algebraic Effects and Handlers. In R. Heckel and S. Milius, editors, *Algebra and Coalgebra in Computer Science*, page 1–16, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40206-7. doi: 10.1007/978-3-642-40206-7_1. URL `https://doi.org/10.1007/978-3-642-40206-7_1`.

G. Bernardi, O. Dardha, S. J. Gay, and D. Kouzapas. On duality relations for session types. In M. Maffei and E. Tuosto, editors, *Trustworthy Global Computing*, pages 51–66, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45917-1. doi: 10.1007/978-3-662-45917-1_4. URL `https://doi.org/10.1007/978-3-662-45917-1_4`.

J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), Nov. 2020. doi: 10.1145/3428194. URL `https://doi.org/10.1145/3428194`.

J. Carette, O. Kiselyov, and C.-c. Shan. Finally Tagless, Partially Evaluated. In Z. Shao, editor, *Programming Languages and Systems*, pages 222–238, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-76637-7. doi: 10.1007/978-3-540-76637-7_15. URL `https://link.springer.com/chapter/10.1007/978-3-540-76637-7_15`.

J. Cheney and R. Hinze. First-Class Phantom Types. Technical report, Cornell University, 2003. URL `https://hdl.handle.net/1813/5614`.

O. Dardha. Recursive session types revisited. *Electronic Proceedings in Theoretical Computer Science*, 162:27–34, Aug. 2014. doi: 10.4204/eptcs.162.4. URL `https://doi.org/10.4204%2Feptcs.162.4`.

R. A. Eisenberg and S. Weirich. Dependently Typed Programming with Singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2012. doi: 10.1145/2430532.2364522. URL `https://dl.acm.org/doi/abs/10.1145/2430532.2364522`.

K. Imai, S. Yuen, and K. Agusa. Session Type Inference in Haskell. In K. Honda and A. Mycroft, editors, *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-centric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010*, volume 69 of *EPTCS*, pages 74–91, 2010. doi: 10.4204/EPTCS.69.6. URL `https://doi.org/10.4204/EPTCS.69.6`.

O. Kammar, S. Lindley, and N. Oury. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 145–158, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323260. doi: 10.1145/2500365.2500590. URL `https://doi.org/10.1145/2500365.2500590`.

A. King. With scoped effects, handlers must be a part of the program, 2020. URL `https://gist.github.com/lexi-lambda/d8fe82b2932e77b178d67cac81d0aaee`. Accessed: 2023-10-15.

D. J. King and P. Wadler. Combining Monads. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, pages 134–143, London, 1993. Springer London. ISBN 978-1-4471-3215-8.

O. Kiselyov and H. Ishii. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, page 94–105, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338080. doi: 10.1145/2804302.2804319. URL `https://doi.org/10.1145/2804302.2804319`.

O. Kiselyov, A. Sabry, and C. Swords. Extensible Effects: An Alternative to Monad Transformers. *ACM SIGPLAN Notices*, 48(12):59–70, Sept. 2013. ISSN 0362-1340. doi: 10.1145/2578854.2503791. URL `https://doi.org/10.1145/2578854.2503791`.

D. Leijen. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science*, 153:100–126, June 2014. doi: 10.4204/EPTCS.153.8. URL `https://doi.org/10.4204/EPTCS.153.8`.

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*, POPL '95, page 333–343, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199528. URL `https://doi.org/10.1145/199448.199528`.

S. Marlow et al. Haskell 2010 language report. 2010. URL `https://www.haskell.org/onlinereport/haskell2010/`.

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90008-4. URL `https://www.sciencedirect.com/science/article/pii/0890540192900084`.

M. Neubauer and P. Thiemann. An Implementation of Session Types. In B. Jayaraman, editor, *Practical Aspects of Declarative Languages*, pages 56–70, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24836-1. doi: 10.1007/978-3-540-24836-1_5. URL `https://doi.org/10.1007/978-3-540-24836-1_5`.

D. Orchard and N. Yoshida. Effects as Sessions, Sessions as Effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 568–581, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837634. URL `https://doi.org/10.1145/2837614.2837634`.

D. Orchard, P. Wadler, and H. Eades. Unifying graded and parameterised monads. *Electronic Proceedings in Theoretical Computer Science*, 317:18–38, May 2020. doi: 10.4204/eptcs.317.2. URL `https://doi.org/10.4204%2Feptcs.317.2`.

S. Peyton Jones, K. Hammond, W. Partain, P. Wadler, and C. Hall. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC, 1993. URL `https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler-a-technical-overview/`.

M. Piróg, T. Schrijvers, N. Wu, and M. Jaskelioff. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 809–818, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355834. doi: 10.1145/3209108.3209166. URL `https://doi.org/10.1145/3209108.3209166`.

G. Plotkin and J. Power. Algebraic Operations and Generic Effects. *Applied Categorical Structures*, 11(1):69–94, Feb. 2003. ISSN 1572-9095. doi: 10.1023/A:1023064908962. URL `https://doi.org/10.1023/A:1023064908962`.

G. Plotkin and M. Pretnar. Handlers of Algebraic Effects. In G. Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-00590-9.

R. Pucella and J. A. Tov. Haskell Session Types with (Almost) No Class. *SIGPLAN Notices*, 44(2):25–36, Sept. 2008. ISSN 0362-1340. doi: 10.1145/1543134.1411290. URL `https://doi.org/10.1145/1543134.1411290`.

H. Punchihewa and N. Wu. Safe Mutation with Algebraic Effects. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, page 122–135, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386159. doi: 10.1145/3471874.3472988. URL `https://doi.org/10.1145/3471874.3472988`.

T. Schrijvers, M. Piróg, N. Wu, and M. Jaskelioff. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 98–113, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342595. URL `https://doi.org/10.1145/3331545.3342595`.

W. Swierstra. Data Types à La Carte. *Journal Functional Programming*, 18(4):423–436, July 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758. URL `https://doi.org/10.1017/S0956796808006758`.

K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48477-6.

B. van den Berg and T. Schrijvers. A Framework for Higher-Order Effects & Handlers, 2023. URL `https://doi.org/10.48550/arXiv.2302.01415`.

J. Voigtländer. Asymptotic Improvement of Computations over Free Monads. In P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction*, pages 388–403, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70594-9. doi: 10.1007/978-3-540-70594-9_20. URL `https://doi.org/10.1007/978-3-540-70594-9_20`.

N. Wu and T. Schrijvers. Fusion for Free - Efficient Algebraic Effect Handlers. In R. Hinze and J. Voigtländer, editors, *Mathematics of Program Construction*, pages 302–322, Cham, 2015. Springer International Publishing. doi: 10.1007/978-3-319-19797-5_15. URL `https://doi.org/10.1007/978-3-319-19797-5_15`.

N. Wu, T. Schrijvers, and R. Hinze. Effect Handlers in Scope. *ACM SIGPLAN Notices*, 49(12):1–12, Sept. 2014. ISSN 0362-1340. doi: 10.1145/2775050.2633358. URL `https://doi.org/10.1145/2775050.2633358`.

N. Xie and D. Leijen. Effect Handlers in Haskell, Evidently. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, Haskell 2020, page 95–108, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380508. doi: 10.1145/3406088.3409022. URL `https://doi.org/10.1145/3406088.3409022`.

Appendix

```haskell
data S a; data O a b ; data SL a
data R a; data C a b ; data CL a

type family Dual' proc
type instance Dual' (R a) = S a
type instance Dual' (S a) = R a
type instance Dual' (O a b) = C (Dual a) (Dual b)
type instance Dual' (C a b) = O (Dual a) (Dual b)
type instance Dual' (CL a) = SL (Dual a)
type instance Dual' (SL a) = CL (Dual a)

type family Dual proc where
  Dual '[] = '[]
  Dual (x : xs) = Dual' x : Dual xs

data Session p q r where
  Send :: a -> Session (S a : p) p ()
  Recv :: Session (R a : p) p a

type End :: [Type]
type End = '[]

data instance ScopeE Session m p p' q' q x' x where
  Offer ::
    m a End x ->
    m b End x ->
    ScopeE Session m (O a b : c) '[O a b] End c x x
  Sel1 ::
    m a End x ->
    ScopeE Session m (C a b : c) a End c x x
  Sel2 ::
    m b End x ->
    ScopeE Session m (C a b : c) b End c x x
  ServerLoop ::
    m a End (Maybe x) ->
    ScopeE Session m (SL a : c) a End c (Maybe x) [x]
  ClientLoop ::
    m a End x ->
    ScopeE Session m (CL a : r) a End r x [x]
```

Listing 1: Full definition of the **Session** effect.

```haskell
send ::
  forall a f p.
  a ->
  PrEff f Session (S a : p) p ()
send a = sendP (Send a)

recv ::
  forall a p f.
  PrEff f Session (R a : p) p a
recv = sendP Recv

sel1 ::
  PrEff f Session a End x ->
  PrEff f Session (C a b : p) p x
sel1 act = sendScoped (Sel1 act)

sel2 ::
  PrEff f Session b End x ->
  PrEff f Session (C a b : p) p x
sel2 act = sendScoped (Sel2 act)

offer ::
  PrEff f Session a End x ->
  PrEff f Session b End x ->
  PrEff f Session (O a b : p) p x
offer s1 s2 = sendScoped (Offer s1 s2)

loopS ::
  PrEff effs Session a End (Maybe x) ->
  PrEff effs Session (SL a : p) p [x]
loopS act = sendScoped (ServerLoop act)

loopC ::
  PrEff f Session a End x ->
  PrEff f Session (CL a : p) p [x]
loopC act = sendScoped (ClientLoop act)
```

Listing 2: Embedding functions for the `Session` effect.

```haskell
connect :: (Dual p1 ~ p2, Dual p2 ~ p1) =>
  PrEff f Session p1 '[] a -> PrEff f Session p2 '[] b ->
  PrEff f IVoid () () (a, b)
connect (Value x) (Value y) = pure (x, y)
connect (ImpureP (Recv) k1) (ImpureP ((Send a)) k2) =
  connect (runIKleisli k1 a) (runIKleisli k2 ())
connect (ImpureP ((Send a)) k1) (ImpureP (Recv) k2) =
  connect (runIKleisli k1 ()) (runIKleisli k2 a)
connect (ScopedP op1 k1) (ScopedP op2 k2) = case (op1, op2) of
   (Sel1 act1, Offer act2 _) -> Ix.do
     (a, b) <- connect act1 act2
     connect (runIKleisli k1 a) (runIKleisli k2 b)
   (Sel2 act1, Offer _ act2) -> Ix.do
     (a, b) <- connect act1 act2
     connect (runIKleisli k1 a) (runIKleisli k2 b)
   (Offer act1 _, Sel1 act2) -> Ix.do
     (a, b) <- connect act1 act2
     connect (runIKleisli k1 a) (runIKleisli k2 b)
   (Offer _ act1, Sel2 act2) -> Ix.do
     (a, b) <- connect act1 act2
     connect (runIKleisli k1 a) (runIKleisli k2 b)
   (ServerLoop act1, ClientLoop act2) -> Ix.do
     (a, b) <- connectLoop act1 act2
     connect (runIKleisli k1 a) (runIKleisli k2 b)
   (ClientLoop act1, ServerLoop act2) -> Ix.do
     (a, b) <- connectLoop act2 act1
     connect (runIKleisli k1 b) (runIKleisli k2 a)
 where
  connectLoop bodyA bodyB = go ([], [])
   where
     go (r1, r2) = Ix.do
       (a, b) <- connect bodyA bodyB
       case a of
         Nothing -> pure (r1, b : r2)
         Just a' -> go (a' : r1, b : r2)
connect (Impure cmd k1) k2 = Impure cmd $ iKleisli $
  \x -> connect (runIKleisli k1 x) k2
connect k1 (Impure cmd k2) = Impure cmd $ iKleisli $
  \x -> connect k1 (runIKleisli k2 x)
```

Listing 3: Algebraic effect handler for the **Session** effect that interprets two procedures using cooperative multitasking.