Eberhard Karls Universität Tübingen
Faculty of Mathematics and Science
Department of Computer Science

Master's Thesis

Second-class modules for the Effekt programming language

Roman Schulte

September 30, 2021

Reviewers

Prof. Dr. Klaus Ostermann
Programming Languages
Department of Computer Science
Eberhard Karls Universität Tübingen

Prof. Dr. Torsten Grust
Database Systems
Department of Computer Science
Eberhard Karls Universität Tübingen

Advisors

Dr. Jonathan Brachthäuser
Programming Methods Laboratory
École Polytechnique Fédérale de Lausanne

Philipp Schuster
Programming Languages
Department of Computer Science
Eberhard Karls Universität Tübingen

Schulte, Roman:
Second-class modules for the Effekt programming language
Master Thesis Computer Science
Eberhard Karls Universität Tübingen
Thesis period: 01.04.2021 - 30.09.2021

Abstract

Effekt is a research programming language that features a lightweight implementation of effects and handlers. As part of this thesis, the language was extended with second-class modules and related functionalities. These new features allow programmers to define modules and abstract them with interfaces. Also, the module system aims to strengthen the interplay between effects and modules. Both concepts share common characteristics, paving the way for potential future unification. This work examines those possibilities and provides insights into the relationship of modules and effects.

Zusammenfassung

Die steigende Komplexität von Computerprogrammen stellt Entwickler vor ein großes Problem. Sie müssen komplizierten Quelltext verstehen und die Verbindungen zwischen den einzelnen Komponenten verinnerlichen, um bei Änderungen nicht das System aus dem Gleichgewicht zu bringen. Als Abhilfe für dieses Problem bieten sich die Modularisierung von Anwendungen an. Diese Technik basiert auf dem Konzept von Modulen, welche die Bausteine für Programme bilden. Programmiersprachen können um diese Konstrukte erweitert werden, um die Anwendung von Modulen zu vereinfachen.

Module alleine können jedoch nicht jedes Problem beheben, welches komplexe Software mit sich bringt. Ein weiterer Stolperstein entsteht durch das Auftreten von Seiteneffekten. Als Seiteneffekt bezeichnet man zusätzliche Auswirkungen auf den Zustand eines Programms, welche während dem Ausführen von Funktionen auftreten. Auch hier gibt es eine bekannte Lösungsmöglichkeit: Algebraische Effekte. Diese helfen Programmieren das Auftreten von Seiteneffekten klar zu spezifizieren und zu kontrollieren.

Effekt ist eine Programmiersprache welche aus der Forschung an Algebraischen Effekten hervorgegangen ist. Im Rahmen dieser Arbeit wurde nun ein Modulsystem in diese Sprache integriert. Die Motivation für diese Erweiterung ist die Untersuchung des Zusammenspiels zwischen Effekten und Modulen. Als Ergebnis wurden Gemeinsamkeiten und Zusammenhänge beider Konzepte entdeckt, welche Grundlage sein können für weitere Forschung auf diesem Gebiet.

Contents

List of Figures v

1 Introduction 1

2 Concepts 3

2.1 Modules . . . 3

2.1.1 Modular Programming . . . 3

2.1.2 Separate Compilation . . . 5

2.1.3 Information Hiding . . . 6

2.1.4 Packages and Namespaces . . . 7

2.2 Algebraic Effects . . . 8

2.2.1 Side-Effects . . . 8

2.2.2 Effectful Functions . . . 9

2.2.3 Effect Handlers . . . 9

2.2.4 Exceptions and Continuations . . . 10

2.3 Contribution . . . 11

2.3.1 Main Goal . . . 12

2.3.2 Key Challenges . . . 12

3 Application **15**

3.1 Basic Usage . . . 15

3.1.1 Hello World . . . 15

3.1.2 Nested Modules . . . 16

3.1.3 Importing Modules . . . 17

3.2 Abstraction 18
3.2.1 Module Interfaces 18
3.2.2 Top-Level Implementation 19
3.3 Effectful Programming 20
3.3.1 Effects vs Interfaces 21
3.3.2 Modules as Handlers 22

4 Design 25
4.1 Design Decisions 25
4.1.1 First-Class vs. Second-Class Modules 25
4.1.2 Nominal vs Structural Typing 26
4.1.3 Stateful Modules 28
4.2 Further Improvements 28
4.2.1 Unifying Modules and Capabilities 28
4.2.2 Local Modules 29

5 Implementation 33
5.1 Compilation Pipeline 33
5.1.1 Parser 34
5.1.2 Frontend 34
5.1.3 Backend 35
5.1.4 Generator 35
5.2 Symbols 37
5.2.1 Module Type 37
5.2.2 Module Symbol 38
5.2.3 Source Module 39
5.2.4 User Module 39
5.2.5 Module Parameter 40

6 Conclusion 41

Bibliography 43

List of Figures

2.1 Example of a module dependency graph. 5
2.2 Example of an effect definition in the Effekt programming language. 8
2.3 Function that uses the effect `Write` from Figure 2.2. 9
2.4 Effect handler for Figure 2.3 that resumes the function. 10
2.5 Effect handler for Figure 2.3 that provides a result. 10

3.1 Hello world program written in Effekt using modules. 16
3.2 Example of nested modules. 17
3.3 Importing source modules. 17
3.4 Definition, implementation and usage of a module interface. . 19
3.5 Source module implementing an interface using top-level definitions. 20
3.6 Usage of an effect to model the same logic from Figure 3.4. . . 21
3.7 Using a module interface to handle effects. 23

4.1 Pseudo-code for modules in a structural (a) and nominal (b) typed setting. 27
4.2 Pseudo code featuring a syntax to use module as handlers. . . 29
4.3 Pseudo code for the possible usage of local modules. 30

5.1 Simplified compiliation pipeline. 33
5.2 Simplified AST representing the example from Figure 3.4. . . 34
5.3 JavaScript code generated from Figure 3.4 by the compiler . . 36
5.4 Simplified class-hierarchy of module-related symbols. 37

Chapter 1

Introduction

A major challenge in modern software development is the increasingly complexity of programs. This problem affects developers as well as their tools. Programmers need to overlook extensive code bases and remember complex interactions between various components. Also, their tool chain might suffer from this burden because even tiny changes require the entire project to be rebuild. This costs valuable time and computing resources. A common solution to those problems is modularization. This technique involves fine-grained software entities called *modules*. The goal of this process is the decomposition of complex systems into manageable and easily understandable units. This allows programmers to unravel monolithic applications and delegate responsibilities to certain modules. Developer tools like compilers can benefit from this transition too. They can track dependencies between modules and only recompile them if needed. Thus, it is desirable for programming languages to implement a module system.

However, modules alone cannot remedy all hurdles of complicated software. Namely, side-effects remain an unseen source of errors. A side-effect occurs when a function modifies the execution context as a byproduct of its computation. Such an interference potentially comprises the correctness of an application but can be difficult to detect. Research on this topic culminated in the specification of *algebraic effects*. This allows programmers to be upfront about possible side-effects of a function. Algebraic effects provide a tool to model a side-effect and annotate it to the signature of functions. Calling such an effectful function requires the context to provide handlers for all listed effects. Programming languages with an effect system support the controlled use of side-effects.

Effekt is a functional programming language which supports the definition and handling of effects. Besides type-safety, this language promises to ensure an additional property named *effect safety*. Thus, the compiler checks if

effects are properly annotated to functions and eventually handled. Now, the language is extended with a module system to make it suitable for programming in the large. Interestingly, modules in Effekt are not first-class but second-class entities. This choice is rooted in an effort to pursue compability between effects and modules. Nevertheless, the module system presented in this work can still express common use-cases of modules.

This thesis covers the design and implementation of a module system for the programming language Effekt. It is structured as followed: Chapter 2 presents the concepts of modules and algebraic effects. Those theoretical constructs are complemented with practical examples in the following Chapter 3. Motivated by these uses-cases, Chapter 4 dives into the design of the module system. Chapter 5 briefly walks through the implementation of modules in the Effekt compiler. Lastly, Chapter 6 summarizes the results of this work.

Chapter 2

Concepts

This chapter introduces major concepts from the topics of modules and algebraic effects. The novelty of this thesis grounds on the integration of a second-class module system into an effectful programming language. Both concepts by itself are neither new nor will be reinvented in this work. Modules (Sec. 2.1) and algebraic effects (Sec. 2.2) have been around for quite a while. However, the niche that is explored in this thesis is the combination of both (Sec. 2.3). Before the resulting module system is put into action (Chp. 3), the following sections will provide a brief insight into the worlds of effects and modules.

2.1 Modules

Modules are fundamental building blocks of software and can be combined and composed to create complex systems. Programmers can leverage modules to break monolithic applications into smaller, manageable components (Sec. 2.1.1). This process is called modularization and might benefit users, as well as the compiler itself (Sec 2.1.2). Information hiding (Sec. 2.1.3) has emerges as a key strategy to guide the design of modular applications. It revolves around the idea to use modules to conceal as much implementation details as possible. Compared to other language constructs, like namespaces and packages (Sec. 2.1.4), modules can act as more than simple code containers. They are a versatile tool that can adopt to many use-cases.

2.1.1 Modular Programming

Modular programming aims to make programs flexible and maintainable. Without the usage of modules the code base of a program becomes a

monolith. In this scenario all pieces of the software are tightly-coupled together. This poses a major hurdle for programmers as well as the compiler. Maintainers of such a code base need a deep understanding of the whole program in order to introduce non-breaking changes. Even slight modifications require the compiler to reprocess the entire project. Those problems can be countered with modular programming[6]. Developers use this technique to break programs into smaller parts. Those fine-grained software components are called modules[19]. Each module has a well-defined boundary and relationship to other modules. Programmers can then construct applications by combining various modules, each bearing responsibility for a specific task. Modular programming emphasizes the design of software as a composition of smaller, interchangeable units.

The key benefits of modules are: *lose-coupling*, *separation of concerns*, *code reuse* and *encapsulation*. First off, modularization makes inter-code relationships more visible, since modules clearly define their dependencies towards each other. In most cases, a module might only depend on a general description of a functionality, rather than a specific implementation and thus accounts for lose-coupling. Programmers aggregate related code in one module and distribute responsibilities for certain features of the application over distinct modules. This allows for separation of concerns and efficient code reuse. And last but not least, modules have the ability to hide parts of their implementation from the outside through encapsulation. This is especially handy to protect the internals of a module from unwanted access and/or to guard strict life-cycle protocols, e.g. for files managed by the module. Modules come with various benefits that help programmers to tackle complexity in software design.

Modules exists in various shapes and forms. The Standard ML programming language defines modules as a pair of *signature* and *structure*[13]. In this context, those terms roughly translate to *interface* and *implementation* respectively. Users define signatures and structures separately and provide different implementations for a signature. Also, one structure might implement multiple signatures at once. Programmers dynamically compose modules using *functors*. A functor can be described as a high-level function that takes modules as arguments and produces a new module. Structures, signatures and functors are the cornerstone of ML's module system and can also be found in Wyvern[15], an object-oriented programming language featuring effects (Sec 2.2). Wyvern further divides modules into two categories: *pure* and *resource* modules[14]. The major difference between those two is that a resource module might bear state and can use other resource modules, which is forbidden for a pure module. While there are commonalities between module systems, they differ in crucial details, making each one standing out in their own way.

Understanding software as a network of collaborating modules is the core essence of modular programming. Just like individuals share labor in a collective production, modules work together and distribute responsibility. Each module contributes domain-specific knowledge to the entirety of the system. Multiple modules can provide distinct solutions for the same problem, allowing users to try different strategies. This interchangeability of modules comes from the distinction between the interface of a module and its implementation.

2.1.2 Separate Compilation

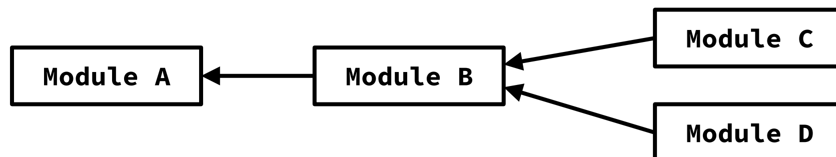

Figure 2.1: Example of a module dependency graph.

Users might want to compile only a subset of modules instead of the entire project. This requires the compiler to track the dependencies of any module. Fig 2.1 depicts such knowledge in form of a dependency graph. Each box in the graph represents a module and an arrow from X to Y indicates that module X depends on Y. In this example, the modules C and D depend on B, while B only depends on A. Therefore, module A is a transitive dependency of C and D. Thus, in order to compile module C, the compiler needs to process A first, continue with B and finally look at the sources of C. In this case, module D does not need to be compiled at all. If B changes afterwards, the compiler only needs to recompile the modules B and C, because the output of A can be reused from the previous run. This technique can have a major impact on the overall compile time of a program.

Separate compilation requires the compiler to produce a deterministic output for any given module. In this context, compiling the same modules twice in a row must lead to an identical result. Only if the compile satisfies this requirement, it might omit additional runs and reuse an existing output. *Cut-off compilation*[1] can further reduce the costs of recompilation by analyzing the signature of modules. In cases where a module changes its implementation but maintains a stable interface, consumers of the module

also do not need to be recompiled. Introducing modules into a programming language also opens the door for additional compiler optimizations.

The benefits of modules are not limited to the users but also affect the compiler as well. Key features of modules include their well-defined boundaries and dependencies. This helps the compiler to understand the structure of a program. The knowledge about these relationships allows to determine an order in which modules need to be compiled. Also, the impact of modifications in the code base can be tracked. The compiler can then decide if a module is affected by those changes and therefore must be recompiled. Due to modules, compilers can make efficient use of computing resources.

2.1.3 Information Hiding

In 1972, Parnas et al.[18] conducted a case study to contrast two approaches of modularization. A simple text processing application serves as starting point for the experiment. Then, the authors present two modular designs centered around different criteria. The first design is derived from a flow-chart representation of the application logic. Each step of the processing pipeline is delegated to a own module. The second approach follows a philosophy known as *information hiding*. In this case, each module encapsulates a design decision, e.g. how a single line is stored. Subsequently, the adaptability of both designs is discussed. For this purpose, possible changes in the implementation details of the application are examined, e.g. supporting a different input format. As a result, the authors conclude that modules should be used to capture and conceal difficult design decisions rather than representing steps of the application flow.

Modules are no silver bullet. Their mere existence does not guarantee programs to be flexible and adaptable. In the worst case, a naive use of modules can even work against those goals. Thus, it is important to chose an appropriate criteria to ground modularization. As a first step towards the decomposition of a system, developers should identify crucial design decisions. Then, each decision can be concealed by a module leaving room for future reconsideration. The modularization of software requires careful planning.

Information hiding postulates the design of modules around implementation details that are likely to change. This principal bases on the distinction between module interfaces and implementations. An interface specifies the general behavior and abstracts the module. For instance, an interface might describe some kind of persistent storage by defining methods to read and write data. Users of this functionality can write code against the interface specification rather than depending on a

certain implementation. Programmers might want to change the location where the data is stored or what format is used in future. Nevertheless, the consumers of this module remain unaffected from this modifications as long as the interface stays stable. Information hiding aims to minimize the overall impact of internal changes on the code base.

Information hiding is closely associated with the separation of concerns and encapsulation. Design decisions belong to those concerns modules are trying to separate. They do so by encapsulating their implementations and shield internals from the outside. Thus, developers can utilize these benefits of modules to achieve flexibility and counter complexity. Programmers can treat information hiding has a guideline to make best use of modules.

2.1.4 Packages and Namespaces

Programming languages like Java[9] and C++[8] use packages or namespaces to organize code. Each package or namespace provide a fresh environment for names. Both concepts are comparable to folders of a file system. They can be recursively nested and their contents can be globally identified using their absolute path, which is also called a *qualified name*. Multiple definitions with the same name and signature can coexist in a project when they are declared in different scopes. Usually these declarations can be referenced in two ways: Either by using the fully qualified name or by importing the package/namespace into the current scope. Thus packages and namespaces divide the code base into smaller, reusable groups.

One might argue that information hiding can be achieved using Java packages, which is not the case. A definition might be prefixed with the `package` keyword. This restricts the access to members of the same package. Therefore this keyword could be used to hide information from the outside and makes it only visible to the inside. However, this argument quickly falls apart with a closer look on how packages are declared. There is no restriction which classes or interfaces can become part of a package. Even if the packages was declared in another library or dependency, the user can sneak in additional definitions. This allows programmers to bypass this kind of access control. Packages in Java are no sufficient tool for information hiding.

Namespaces and packages are neither first- nor second-class objects (Sec. 4.1.1). They mainly exist to organize code and to avoid naming clashes. Their names can only appear as prefix of an identifier to form a qualified name or as part of an `import`/`use` statement. Modules on the other hand have actual instances that can be part of computations. This makes packages and namespaces less powerful than modules. They mainly exist from a structural

point of view. Packages and namespaces cover only a small subset from the features of modules.

2.2 Algebraic Effects

Algebraic effects and handlers offer a concept to control side-effects in programs[22][20][23]. A side-effect is any kind of contextual modification occurring during one function call. Programmers use algebraic effects to model those operations (Sec. 2.2.1). Functions express their side-effects with additional annotations in their signature (Sec. 2.2.2). In order to call such an effectful functions, the calling context has to provide handlers for all declared effects (Sec. 2.2.3). This allows programming languages to ensure that all side-effects are eventually handled, resulting in a property called *effect safety*. Thus, programmers can define, use and handle side-effects in a transparent fashion.

2.2.1 Side-Effects

Computations can have additional byproducts, such as the modification of global state. Traditionally, the signatures of functions express what kind of data is needed to call the function and what can be expected from its return value. However, a function call might also impact the state of the system. Besides from simply computing a result, functions can read and change the value of a global variable or print messages to the console. Those operations are called *side-effects*[21]. They can cause consecutive calls of the same function to yield different results because its execution context changed. Thus, side-effects can have far-reaching consequence on the correctness of a program. This places a burden on consumers of such functions, since their behavior is hard to grasp from the outside. Without further language features, programmers need to inspect the implementation of functions to uncover their side-effects.

```
1 effect Write(msg: String): Unit
```

Figure 2.2: Example of an effect definition in the Effekt programming language.

Programming languages require a way to define side-effects, in order to track them. Figure 2.2 shows an example of a side-effect definition written in Effekt[4]. The keyword `effect` is used to declare a new side-effect with the

name `Write`. This effect consists of a single operation, expecting one argument of type `String` and producing a void result. Thus, this effect represents some kind of output that swallows a textual message. In Effekt, such definition is called an *effect signature*. Programmers can use them to model side-effects.

Effect signatures relate to module interfaces. Both are separated from their implementation and list available operations. An effect signature allows the abstraction of a side-effect with the same methods interfaces use to abstract behavior or data. Thus, effect signatures are the specifications of side-effects.

2.2.2 Effectful Functions

```
1 def hello(name: String): String / { Write } = {
2     var msg = "Hello " ++ name
3     do Write(msg)
4     return msg
5 }
```

Figure 2.3: Function that uses the effect `Write` from Figure 2.2.

The signatures of effectful functions communicate their use of side-effects. Figure 2.3 demonstrates the declaration of such a function. This signature reads as: "Given an argument of type `String`, this function produces a result of type `String` and requires the calling context to handle the effect `Write`."[4]. Inside the body, functions can invoke the operations of those effects with the `do` keyword. In this example, the function `hello` concatenates the string literal `"Hello "` with the input parameter. Then, this message is used to call the effect operation `Write` and finally returned as result value. A function might only use side-effects annotated to its signatures.

Effectful functions are the consumers of effects. This corresponds to the concept of consuming a module. However, instead of declaring a parameter bound to an effect type, programmers annotate the types of required effects to the return type of a function. In Effekt, this is done using the pattern `/ { Effect1, ..., EffectN }`, which describes a set of effects. Also, it is possible to omit this annotation and let the compiler infer the required types. A function consumes effects exclusively through their signatures.

2.2.3 Effect Handlers

Effect handlers provide a mechanism to implement effects. Figure 2.4 and 2.5 both show possible handlers for the effectful function from Figure 2.3.

```
1 def handle1() = {
2     try { var str = hello("Effekt") }
3     with Write { (msg) => println(msg); resume(()) }
4 }
```

Figure 2.4: Effect handler for Figure 2.3 that resumes the function.

```
1 def handle2() = {
2     try { var str = hello("Effekt") }
3     with Write { (msg) => "Cancel" }
4 }
```

Figure 2.5: Effect handler for Figure 2.3 that provides a result.

The first implementation prints the message to the console. After that, the keyword `resume` is used to jump back into the function. Since `Write` declares `Unit` as its return type, the matching literal is passed to `resume`. This allows `hello` to complete its computation and the variable `str` stores the value `"Hello Effekt"`. The second example implements a different behavior. Instead of calling `resume`, this handler provides an alternative return value for the effectful function itself. Thus, the execution of `hello` is not continued. The variable `str` will now contain the value `"Cancel"`. Effect handlers can decide whether the call of an effect operation returns a result or terminates the function.

Effect handlers are the counterpart to effects and provide their implementation. Again, there is a strong analogy to the concept of modules. Compared to a module implementation, handlers have additional abilities, which allow them to cancel the execution of a function. This resembles the control-flow construct of exceptions, which in fact can be treated as simple examples of handlers. An effect handler inverts the control over side-effects from a function to its caller.

2.2.4 Exceptions and Continuations

Exceptions[2] and continuations[7] can alter the execution flow of a program. In the imperative world each function call is executed synchronously and will eventually produce a return value. In some cases developers need to break these constraints. Asynchronous communications like network or bluetooth connections are hard to model using only synchronous functions. If the

program waits for a response, the execution needs to be blocked, which wastes computing resources and makes the application unresponsive. Also, such a communication might be error-prone due to unstable connections or corrupted messages. In this case, functions need to return distinct values to express the failure or success of an operation. Additional language features support developers to deal with these problems.

Continuations and exceptions extend the traditional concept of functions and help to model error-prone and asynchronous code. Instead of providing a result, functions can throw an exception. In this case, the execution halts and all local state is lost. If a programmer only wants to temporary interrupt the execution, coroutines and continuations can come handy. Like exceptions, they offer an additional way of manipulating the execution flow. A coroutine still promises to deliver a result but it might do so in an asynchronous fashion. Continuations are used to capture the state of a suspended coroutine and allow to resume their execution. Thus, continuations and exceptions serve different use-cases.

Plotkin et al.[23] identified exceptions as a basic example of algebraic effects. Programming languages differ in the extent they track exceptions: Python or Kotlin do not track them at all, in Swift the return type needs to be annotated with the `throws` keyword to indicate that a call might fail and Java requires the declaration of each type of exception that might occur during execution. Regardless of these differences, all languages introduce a similar construct to handle exceptions. They all use some kind of `try/catch` statement. The `try` keyword is followed by a scope in which programmers can place unsafe code. It is then followed by one or more `catch` statements, each handling a different type of exception. The definition of handlers in the caller scope is nearly identical to the handling of algebraic effects. However, exception handlers have no means of redirecting the control flow back to the function. This is not true for effect handlers, which can decide whether to resume or abort an execution. Thus, algebraic effects and their handlers combine the abilities of continuations and exceptions.

2.3 Contribution

The results of this work are of practical and theoretical nature. Overall, the motivation of this thesis is to narrow the gap between effect and module systems (Sec. 2.3.1). However, such effort faces hurdles on different layers (Sec. 2.3.2). For example, there is no uniform definition that pins down the concept of a module. The following sections outline the scope and pitfalls surrounding the design and implementation of the module system for Effekt.

2.3.1 Main Goal

While researchers have covered many aspects of effect systems, their combination with modules remains a niche topic. Languages like Frank[12], Helium[3] and Koka[11] offer different implementations of algebraic effects and handlers. However, they lack the ability to define modules. Wyvern[15] is one of the few programming languages that features both a module and an effect system[14]. This language follows the paradigm of object-oriented programming and uses first-class modules to model capabilities. In contrast, Effekt is a functional language that uses capabilities to implement lightweight effects[4] . As part of this thesis, a second-class module system was integrated into Effekt. The contributions of this work can be summarized as:

- the design of a module system that accompanies an existing effect system.
- a proof-of-concept implementation of said system in the Effekt programming languages
- exploration of the relationship between effects and modules

This effort aims to acquire new insights into the overlaps of effects and modules, as well as their possible interactions.

So far, Effekt only provided basic constructs named after modules. They were mainly used to organize source files. Therefore the number of modules in a program was directly linked to its number of files. To break this constraint, additional languages entities are introduced. This also includes a notion for interfaces, allowing users to abstract modules. Every new feature was designed to play well with the existing implementations of effects and handlers.

The comparison of effects and modules point out several commonalities. First, both can be abstracted by defining their signatures. Second, users of an effect or module only depend on its signature rather than an actual implementation. And third, the definition of handlers and modules share a common structure. Thus, the concepts of effects and modules offer potential for further unification.

2.3.2 Key Challenges

The first challenge arises from the general concept of modules. There is no single definition of all characteristics that make up a module. The terms modules and modularity are broadly used in literature. As a consequence, module systems tend to vary in their scope and functionalities. Thus, the design of a module system involves many cross-cutting decisions.

Another possible pitfall comes from comparing modules and effects too superficially. Although their definitions share many similarities, they serve different purposes. Modules are useful to abstract behavior while algebraic effects have an impact on the control flow of a program. Also, there are different limitation when it comes to their signature. For instance, effect signatures cannot contain higher-ordered functions, in order to prevent capabilities from escaping. However, module interfaces can define such operations. It is important to keep these different uses-cases in mind.

A more practical problem originates from the compiler itself. The introduction of new language features is always reflected with changes in the compilation pipeline. This includes the handling of new keywords and syntax rules as well as modifications of the internal representation of a program. Modules offer programmers to distribute their definitions over several distinct scopes. Thus, algorithms involving the resolution of names have to consider additional locations. Unfortunately, the original compiler design did not account for these scenarios.

Chapter 3

Application

This chapter supplements the previously described concepts with practical examples. Modules can serve multiple purposes: on a structural level, a module can act as container to organize and reuse code (Sec. 3.1). More advanced applications of modules involve their abstraction using interfaces (Sec. 3.2). Finally, effectful programs can utilize modules to complement effects (Sec. 3.3). The following code examples serve as motivation for the design of Effekt's module system in the next chapter (Chp. 4).

3.1 Basic Usage

The basic usage of modules involves them as units to collect code. Traditionally, a new language feature is introduced with a program simply printing the phrase `"Hello World"` (Sec. 3.1.1). This illustrates the syntax to define modules. Besides from functions, a module can also contain definitions of other modules (Sec. 3.1.2). Finally, code reuse is facilitated through imports (Sec. 3.1.3). Although these examples are rather trivial, they form the foundation for later applications.

3.1.1 Hello World

In their simplest use-case, modules can serve as namespaces. Figure 3.1 shows a hello world program written in Effekt, using the new module features. The first usage of the keyword `module` defines the name of the top-level module, which contains all definitions in the outer scope of the file. Next, the `module` keyword is used again, but this time it defines a scope delimited by `{ }`. All definitions inside those delimiters are members of the module. In this case, `Hello` contains only one member function, defined with the `def` keyword. Finally, there is

```
1  // (1) Source Module
2  module examples/mods
3
4  // (2) User Module
5  module Hello {
6      def world() = {
7          println("Hello World!")
8      }
9  }
10
11 def main() = {
12     // (3) Module Call
13     Hello:world()
14 }
```

Figure 3.1: Hello world program written in Effekt using modules.

a top-level function named `main`, which serves as main entry point for the execution. This function uses the module call syntax (`:`) to interact with the member function `world` from the module `Hello`. Executing the code above will print the familiar phrase `Hello World!` to the console. This example illustrates how users can directly interact with modules.

There are two kinds of modules, named *source modules* and *user modules*. A source module represents the entirety of an input source, e.g. a file. Thus, they serve as container for all top-level definitions. Their names follow a naming scheme borrowed from unix paths and can contain multiple name segments. A user module offers a more lightweight alternative to source modules, because their existence is not linked to a physical file. Thus, user modules come at virtually no cost. Both variants of modules provide a fresh environment for names.

3.1.2 Nested Modules

A module might also contain the definition of other modules. Figure 3.2 shows a nested module. Programmers can use the `module` keyword inside the scope of another modules to define a nested module. In this example, the definition of `Bar` is placed inside the scope of `Foo`, thus making `Bar` a child of `Foo`. `Bar`'s member function `ask` is implemented by delegating the call to `Foo`'s `answer` function. Top-level functions like `main` can chain consecutive module calls to reach members of nested modules. Calling the `main` function of this example

```
1  module examples/nest
2
3  module Foo {
4      def answer(): Int = 42
5
6      module Bar {
7          def ask(): Int = answer()
8      }
9  }
10
11 def main() = {
12     println(Foo:Bar:ask())
13 }
```

Figure 3.2: Example of nested modules.

will write the number 42 to the console. Module definitions can appear on the top-level as well as inside other modules.

Modules can access definitions from their parent scopes. A nested module can directly refer to the members of their parent without using the module call syntax. This resembles the reference of top-level functions, which can also be called without supplying the name of a module. Programmers can reach definitions inside nested modules by chaining the names with the : character. Thus, modules can be indefinitely nested and their members are still reachable from the global scope.

3.1.3 Importing Modules

A source can reuse code from other sources. Programmers can load definitions from another source into the current scope by importing the corresponding source module. Imported modules become dependencies of the current scope. Thus, import statements determine the order in which modules are compiled.

```
1  module examples/a
2
3
4  def answer(): Int = 42
```

(a) Module A

```
1  module examples/b
2  import examples/a
3
4  def ask(): Int = answer() * 2
```

(b) Module B

Figure 3.3: Importing source modules.

Figure 3.3 demonstrates the usage of the import syntax . The left side of the figure contains exemplary code of a small utility module. It defines and implements a single function named `answer`. Module B wants to reuse this definition. Thus, the developer of module B imports module A with the `import` keyword, followed by the qualified name of the module. Afterwards, the implementation of `ask` can call `answer` as part of its computation.

The compiler uses source modules to facilitate separate compilation. A module and one of its dependencies might import the same source. In this case, the shared source is only compiled once. Also, modules that are neither a direct nor transitive dependency are not processed at all. Thus, the compiler treats source modules as fine-grained compilation units.

3.2 Abstraction

Users can abstract modules with interfaces. This mechanism separates the implementation of a module from its specification. Interfaces only declare the signatures of operations. Thus, in order to conform to an interface a module has to provide matching implementations (Sec. 3.2.1). Programmers can use this technique with user modules as well as source modules (Sec. 3.2.2). In the later case, top-level definitions provide the implementation for the interface members. Thus, both kinds of modules can implement interfaces.

3.2.1 Module Interfaces

Interfaces define a set of operations. An interface specifies the signatures of its members. This allows developers to provide abstract descriptions of behavior. Thus, users of an interface do not depend on a specific implementation. Modules and interfaces share a many-to-many relationship. Various implementations of the same interface can coexist in a program and one module might implement multiple interfaces.

Figure 3.4 depicts the definition, implementation and usage of a module interface. Programmers can use the `interface` keyword to define interfaces, like `Worker`. Interface definitions are only permitted on the top-level of a file. The body of such an definitions can contain multiple signatures of operations, declared with `def`. A module can implement interfaces with the `implements` keyword. This obligates the module to provide an implementation for each member of the interface. In this example, the module `Foo` conforms to the interfaces `Worker`. Functions can declare module parameters using the `with` keyword, followed by curly braces containing the names and types of these parameters. The function `work` defines a single module parameter named `mod`,

```
module examples/abstract

interface Worker {
    def bar(): Int
    def baz(): Int
}

module Foo implements Worker {
    def bar(): Int = 40
    def baz(): Int = 2
}

def work() with { mod: Worker }: Int = {
    mod:bar() + mod:baz()
}

def main() = {
    println(work() with Foo)
}
```

Figure 3.4: Definition, implementation and usage of a module interface.

bound to the type `Worker`. Thus, the body of this function can interact with the interface members using the module call syntax. The implementation of `Worker` is selected at the call-side. This is demonstrated in the function `main`. Here, the module `Foo` is passed as an argument to the call of `work` using the `with` keyword. Therefore, this invocation returns the integer `42`, which is then printed to the console.

Interfaces are a crucial tool for information hiding. Rather than calling module members directly, programmers can specify an interface that contains the required operations. Thus, client code can rely on a general description of a functionality. This allows to decouple components and to shield implementation details from the outside. Programmers can use interfaces to establish a boundary between modules and their users.

3.2.2 Top-Level Implementation

Source modules also can implement interfaces. Since top-level definitions are members of a source module, they can be used to fulfill the requirements of an interface. Similar to user modules, the declaration of source modules can have implementation clauses. Thus, interfaces can abstract both kinds of modules.

```
1  module examples/abstract2
2  implements Worker
3
4  interface Worker {
5      def bar(): Int
6      def baz(): Int
7  }
8
9  def bar(): Int = 40
10 def baz(): Int = 2
11
12 def work() with { mod: Worker }: Int = {
13     mod:bar() + mod:baz()
14 }
15
16 def main() = {
17     println(work() with /examples/abstract2)
18 }
```

Figure 3.5: Source module implementing an interface using top-level definitions.

Figure 3.5 depicts a source modules that implements an interface. The workflow follows the same pattern as in the previous example. But this time, the implementations of `bar` and `baz` are declared on the top-level. Consumers, like the function `work`, remain unaffected from these changes. They still depend on the same interface definition. Programmers can pass source modules as arguments by referencing their name, prefixed with an additional `/` character. This prefix is needed to distinguish between the names of user and source modules. The behavior of this example is exactly the same as the one of Figure 3.4.

3.3 Effectful Programming

The following sections demonstrate applications of modules in effectful programs. So far, the previous examples did not use effects at all. Sometimes, an effect signature can be used in the same places as an interface (Sec. 3.3.1). Therefore, it is important to understand which use-cases should favor interfaces over effects and vice versa. But there are also scenarios that require the use of interfaces. For instance, only an interface is allowed to specify operations with

functional parameters (Sec. 3.3.2). This mechanism can be used to switch different handler strategies.

3.3.1 Effects vs Interfaces

In some cases, effects and interfaces can be used interchangeably. The syntax to define an interface or effect follows nearly identical rules. Programmers might face the question which of the two should be used, because the difference between both might narrow down to a single keyword. However, the distinction becomes quite clear when it comes to their implementation. Handlers are defined locally inside functions, while modules are available from the global scope. Also, effect handlers have the ability to alter the control flow, which modules cannot do.

```
1  module examples/eff
2
3  effect Worker {
4      def bar(): Int
5      def baz(): Int
6  }
7
8  def work(): Int / {Worker} = {
9      bar() + baz()
10 }
11
12 def main() = {
13     try { println(work()) }
14     with Worker {
15         def bar() = resume(40)
16         def baz() = resume(2)
17     }
18 }
```

Figure 3.6: Usage of an effect to model the same logic from Figure 3.4.

Figure 3.6 implements the same logic as 3.4, but uses an effect instead of an interface. In this example, the definition of `Worker` now uses the `effect` keyword instead of `interface`. Thus, modules can no longer implement the type `Worker`. The next change occurred in the signature of `work`. Rather than declaring a module parameter, the effect `Worker` is now annotated to the

return type. As a consequence, the member operations `bar` and `baz` can be called directly. Executing this code will still lead to the same result.

In general, programmers should use interfaces to abstract behavior. Calling an effect operation might abort the execution of a function, depending on the handler. For instance, if the implementation of `bar` is changed to `bar() = 40`, executing the program will no longer print the number `42`, but `40`. Thus, an interface forces its implementation to provide an actual result whenever an operation is called. Effects and interfaces communicate different intends.

3.3.2 Modules as Handlers

An interface can declare signatures of higher-ordered functions. This ability is unique to interfaces, because effect operations are forbidden to have functional parameters. They could allow a capability to escape the current scope. Whereas, modules can use this feature to provide handler functions. Such a function is a common pattern in Effekt to facilitate the reuse of handlers. Programmers can use modules and interfaces to specify and provide effect handlers.

Figure 3.7 shows an interface that allows to handle effects through its operations. First, two effects `Add` and `Lit` are defined, to model the computation of terms. The interface `Calc` defines two corresponding operations that take an effectful function as a parameter. Note that, the operations itself declare no effects. Next, the module `Foo` provides an implementation for this interface. The effect handlers are defined inside `handleAdd` and `handleLit`. They are used to handle the effects from their input parameter. Then, the function `work` consumes the interface `Calc` in form of a module parameter. This allows programmers to compose the handlers from the interface. Calling `work` with the implementation of `Foo` leads to the computation of the term `40 + 2` and prints the result `42` to the console.

```
1  module examples/mods/handler
2
3  effect Add(l: Int, r: Int): Int
4  effect Lit(x: Int): Int
5
6  interface Calc {
7      def handleAdd { f: () => Unit / {Add} }: Unit
8      def handleLit { f: () => Unit / {Lit} }: Unit
9  }
10
11 module Foo implements Calc {
12     def handleAdd { f: () => Unit / {Add} }: Unit = {
13         try { f() }
14         with Add { (l, r) => resume(l + r) }
15     }
16
17     def handleLit { f: () => Unit / {Lit} }: Unit = {
18         try { f() }
19         with Lit { (x) => resume(x) }
20     }
21 }
22
23 def work() with { calc: Calc } = {
24     calc:handleLit { calc:handleAdd {
25         println(do Add(do Lit(40), do Lit(2)))
26     }}
27 }
28
29 def main() = {
30     work() with Foo
31 }
```

Figure 3.7: Using a module interface to handle effects.

Chapter 4

Design

This chapter elaborates the design of modules for `Effekt`. The specification of the module system is connected with a series of design decisions (Sec. 4.1). Those shape the strengths and weaknesses of modules in Effekt. Thus, each decision comes with a number of benefits and trade-offs. The design presented in this chapter is by no means final. There is room for further improvements that could consolidate the interactions between effects and modules (Sec, 4.2).

4.1 Design Decisions

This section explores the design space of modules in combination with effects. First, we motivate our pick to make modules second-class (Sec. 4.1.1). Next, we discuss the typing of modules in Effekt (Sec. 4.1.2). This requires to answer the question, when module types are considered compatible. Finally, we examine mutable state in modules and the problems that might arise from it (Sec. 4.1.3). Altogether, the design aims to integrate modules seamlessly with existing language features of Effekt.

4.1.1 First-Class vs. Second-Class Modules

Entities of programming languages are categorized as either first-class or second-class citizens. Strachey[24] coined these terms in the 60's to contrast real numbers and procedures in ALGOL. The categorization of an entity as either first- or second-class expresses in which positions it can be used. In order to become a first-class citizen an object must support to be:

1. passed as argument
2. returned from a function

3. assigned to variables

Usually, the primitive data types like `Boolean`, `Int` and `Double` fulfill these requirements. The usage of second-class entities is more restricted. They can only appear in argument positions. For example, functions in Effekt are second-class, since they cannot be stored in variables. The decision whether a new language construct should be a first- or second-class citizen has a deep impact on its usage.

The above description suggest that first-class citizens are more powerful but the question remains if its desirable to categorize modules as first-class constructs. Osvald et al.[17] pick up this issue. Since first-class entities can be assigned to variables, they might escape their defining scope. This ability becomes a disadvantage when the object represents a capability. Also strict lifecycle protocols, e.g. for files or network connections, become harder to overlook. Second-class entities avoid these problems because they cannot escape their scope. As a consequence, only a second-class entity can capture other second-class entities. Otherwise, first-class objects would be able to leak them. Depending on the use-case it is favorable to deal with second-class rather than first-class objects.

We decided to make modules in Effekt second-class. Capabilities play a huge role in Effekt's lightweight way of handling effects. It comes naturally to design its module system with that fact in mind. Implementing modules as second-class entities brings them closer to capabilities. However, this decision also has its downsides. ML-style functors can no longer exist in this setting, because modules are not permitted as return value. This is a considerable caveat of this approach, since functors are essential constructs in other module systems. In the end, the similarities between modules and capabilities underline the need for both to fall in the same category.

4.1.2 Nominal vs Structural Typing

The next question arises from the compatibility of module types. There are two common approaches to tackle this problem: nominal typing and structural typing. Figure 4.1 depicts pseudo code for both scenarios. The left side (a) demonstrates nominal typed modules. In this setting, users define module types in form of interfaces. They are identified by their name and modules have to explicitly declare what types they intend to implement. Functions also uses these types to declare a module parameter. In this example, both implementations `Foo` and `Bar` could be passed as argument to `work`, because they implement `Worker`. Alternatively, the type of a module can be determined by its structure (b). In this case, a module type is defined by the set of

```
1  interface Worker {
2    def baz(): Int
3  }
4
5  module Foo: Worker {
6    def baz(): Int = 42
7  }
8
9  module Bar: Worker {
10   def baz(): Int = 0
11 }
12
13 def work(mod: Worker)
```

(a) Nominal typed modules

```
1
2
3
4
5  module Foo {
6    def baz(): Int = 42
7  }
8
9  module Bar {
10   def baz(): Int = 0
11 }
12
13 def work(mod: Foo)
```

(b) Structural typed modules

Figure 4.1: Pseudo-code for modules in a structural (a) and nominal (b) typed setting.

member signatures. Thus, the types of `Foo` and `Bar` on the right side would be considered equal. Both contain exactly one member function with the same signature, although their implementations differ. In this scenario, module parameters directly refer to a module, like the parameter `mod` of `work`. Still, `Bar` can be used as argument to call `work`, because its type is equal to the one of `Foo`. Modules types can either be defined explicitly or implicitly.

Structural typing is deemed to be more flexible than the nominal approach[5][10]. In a structural typed setting, programmers are exempt from the burden to maintain separate interface definitions alongside their implementations. Also, retroactive abstraction is easier to achieve. Existing modules can automatically conform to newly introduced types, without touching their implementation. This is especially desirable in cases where the source the of a module is not available to the user. However, this advantage can also turn into a downside. The types of modules could accidentally correspond to each other, even tough they serve different purposes. Also, correctly assigning blame for the missing conformance between two types is a tough challenge in the structural typed setting[16]. Thus, the flexibility surrounding structural types is a double-edged sword.

Modules in Effekt are nominal typed. On the one hand, the explicit use of module types communicates the intend of developers more clearly. On the other hand, this increases the amount of boilerplate code related to modules. However, nominal typed modules fit better in the existing nominal typing for effects.

4.1.3 Stateful Modules

Commonly, modules can bear state. Besides from functions, modules might define variables to store data. In this case, we speak of stateful modules. As already mentioned, altering such state as part of a computation is considered a side-effect. Since modules are globally available in Effekt, this would grant uncontrolled access to their state. Thus, stateful modules could potential introduce additional side-effects.

We decided to forbid mutable state in modules generally. This decision poses a significant drawback to the usability of the module system. One can think of various use-cases that require modules to manage state, e.g. a logger that buffers messages or global counters. For now, those applications are beyond the scope of modules in Effekt. This is a tough limitation of the module system but reversing this choice would come at cost of the integrity of the languages. Allowing users to trigger unspecified side-effects would diminish previous efforts of the effect system. Therefore, modules have to renounce from the use of mutable state in favor of upholding effect-safety.

4.2 Further Improvements

The presented design marks the first iteration of modules in Effekts. As such, the module system is not finalized and can be extended in the future. For instance, the overlaps between module and effect signatures invite to streamline their definitions (Sec. 4.2.1). At least, modules could act as alternative handler implementations. Also, local module definitions could possibly compensate for some use-cases limited by the current design (Sec. 4.2.2). These ideas could serve as starting points for future researches on the topics of modules and effects.

4.2.1 Unifying Modules and Capabilities

Currently, modules cannot implement effect signatures. This might be surprising, since the definition of effect operations is more restricted than their counterparts in interfaces. However, it would be difficult to incorporate the control-flow characteristics of handlers into modules. Nevertheless, a module could still replace a handler that uses the `resume` keyword in all operations. Potentially, modules could serve as capabilities in some cases.

Figure 4.2 depicts a scenario in which a module implements an effect signature and is used as handler. First, this example defines an effect signature named `Worker`. This signature is then implemented by the module `Foo`,

```
effect Worker {
    def bar(): Int
    def baz(): Int
}

module Foo handles Worker {
    def bar(): Int = 40
    def baz(): Int = 2
}

def work(): Int / {Worker} = {
    bar() + baz()
}

def main() = {
    try { println(work()) }
    with Worker by Foo
}
```

Figure 4.2: Pseudo code featuring a syntax to use module as handlers.

illustrated with the fictional keyword `handles`. This module defines two member functions, which provide implementations for the effect operations. The top-level function `work` serves as an user of the effect. Finally, `main` tries to call `work`. Thus, it must provide a handler for the effect `Worker`. This is done by referencing the module `Foo`, using another new keyword `by`. The presented program is intended to be logically equivalent to the example from Figure 3.6. Therefore, modules could act as handlers that always resume a function.

The pictured technique swaps conceptual components from effects and modules. Both concepts are divided into three major parts: their signature, implementation and consumers. The example above combines a specification from the world of effects with an implementation from the universe of modules. This facilitates an interesting new mixture between modules and effects.

4.2.2 Local Modules

The current design does not account for the definition of local modules. Module implementations can only occur outside of functions. This makes them visible to all parts of the program. Programmers might want to limit the access to modules or close their definition over local variables or even capabilities.

In fact, the syntactical structure of handlers is already close to the one of modules. Theoretically, there is nothing that would prevent the introduction of local modules.

```
effect State {
    def get(): Int
    def set(val: Int): Unit
}

interface Counter {
    def next(): Int
    def reset(): Unit
}

def work() with { c: Counter } = ...

def main() = {
    var state = 0

    try {
        work() with Counter { // Local module
            def next(): Int = {
                val n = do get()
                do set(n + 1)
                return n
            }

            def reset(): Unit = {
                do set(0)
            }
        }
    } with State { // Handler
        def get() = resume(state)
        def set(n: Int) = { state = n; resume(()) }
    }
}
```

Figure 4.3: Pseudo code for the possible usage of local modules.

Figure 4.3 depicts a possible syntax and use-case for local modules. This example begins with the definition of an effect `State` and an interface `Counter`. They are followed by a function with a module parameter and an arbitrary implementation. So far, the existing language features are sufficient to implement that logic. This changes when it comes to the function `main`. The

`try/with` statement is used to provide an handler for the effect `State`. This facilitates a capability in the scope of `try`, which allows to call the operations defined in `State`. However, `work` requires a module parameter but not an effect. Instead of passing an existing implementation of the interface `Counter` to `work`, a local module is defined. Thus, the `with` keyword is followed by the name of an interface to which the local module should conform. This creates a new scope where the implementation of `Worker` is placed. The implementations of the interface members invoke operations from the `State` effect. Thus, the local module captures the capability introduced by the handler of `State`. This example also highlights the compatibility between modules and capabilities.

Capturing a capability in a local module definition is safe, since both are second-class entities. There is no chance, that the local module might escape the scope. Thus, its definition can safely close over other second-class objects, such as capabilities. Allowing users to combine modules and effects/capabilities opens up new areas of application for both constructs. Implementing local module definitions in Effekts, could potentially make up for the loss of functors and stateful modules.

Chapter 5

Implementation

This chapter summarizes the implementation of modules into the compiler of Effekt. The compilation process is divided into multiple stages. Different phases of the compiler handle distinct tasks, such as type-checking or code generation (Sec. 5.1). Internally the entities of a program are represented with symbols (Sec. 5.2). Thus, the different components of the module system, like interfaces and user-defined modules, correspond to different classes of symbols. The following sections briefly summarize the modifications of the compiler.

5.1 Compilation Pipeline

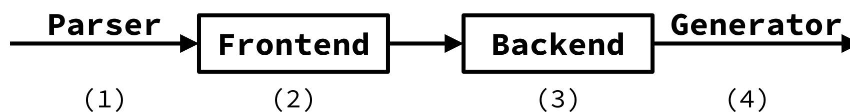

Figure 5.1: Simplified compiliation pipeline.

The pipeline of the Effekt compiler (Fig. 5.1) can be divided into four phases. They start with reading the source files and end with the generation of JavaScript code. Each phase receives the output of the preceding stage and augments it with additional information. The four phases can be summarized as:

1. Parser (Sec. 5.1.1): Reads the source text and transforms it into an Abstract Syntax Tree (AST).

2. Frontend (Sec. 5.1.1):: Receives the AST and creates matching symbols.

3. Backend: Combines information from the AST and symbols to create a core language representation.s
4. Generator: Transforms the core tree into output files, e.g. JavaScript or Scheme code.

The following sections briefly explains each step of the compilation process.

5.1.1 Parser

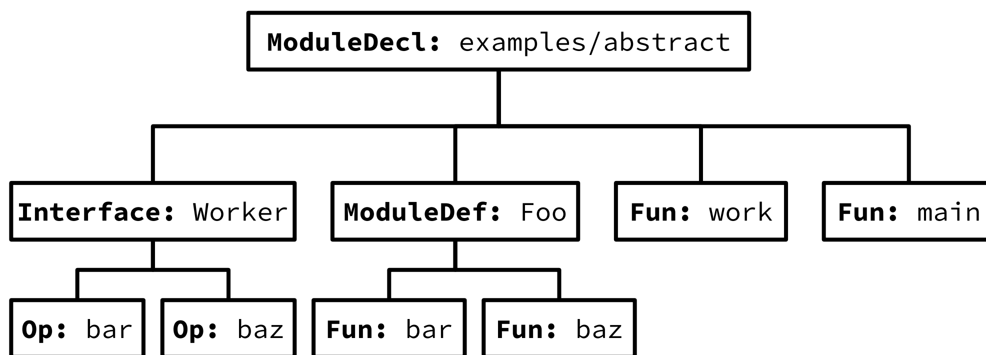

Figure 5.2: Simplified AST representing the example from Figure 3.4.

This phase is responsible for translating the textual representation of a program into a typed data structure. These structures are called *abstract syntax trees* (AST). Figure 5.2 depicts a simplified version of such an AST, representing the source code from Figure 3.4. The root of this tree is the module declaration from the top of the file. This data type holds a list of child nodes, representing the definitions from the top-level. For instance, one of this children is the interface definition of `Worker`, containing the signatures of `bar` and `baz`. The definition of the module `Foo` also contains elements with the name `bar` and `baz`. However, those definitions use a different data type. These types are used to distinguish between the signatures and implementations of functions. This AST supplies the consecutive pipeline with all required information about the input program.

5.1.2 Frontend

The frontend of the compiler process the AST and validates the semantics of a program. A major part of this phase is the `Namer`. This stage of the compiler generates symbols for each entity in the AST, e.g. modules. It

also resolves module calls, like `Foo:bar()`, and references, like module types mentioned in `implements` clauses. Aside of this, the `Namer` phase populates the module symbols, which now contain the types and terms defined in their scope. This information is crucial to the following stage, the `Typer`. At this point, the implementation of a module is type-checked. Thus, the conformance of modules to their declared interfaces is verified. Following phases can lookup the symbols defined in the frontend through a global database.

5.1.3 Backend

The backend phases transform the source AST into a core language representation. This allows the compiler to perform optimizations of the input program. Thus, the core language serves a different purpose than the one used to write the source code. However, these phases of the pipeline remained largely unaffected by the implementation of modules. Solely the core AST was extended by one additional node, representing a user module. At this point, interfaces play no role, because they are not represented in the output language. The result of the backend phase is a core AST, which is handed over to the generator.

5.1.4 Generator

The generator translates the core AST into JavaScript[25] code. Compiling the content of Figure 3.4 produces the output shown in Figure 5.3. The first line imports the compiled standard library and stores its content in `$effekt`. This is exemplary for the translation of module imports. Next, a variable named `$example_abstract` is defined as an empty object. This variable represents the source module and will be populated and exported at the end of the file. User modules, like `Foo`, are computed from an anonymous function, which contains the module members as local definitions. The result of this function is also an object that exports the members. Note, that some definitions are exported under multiple names. This is due to them being used to implement an interface. Module parameters, like the one of `work`, become regular parameters in JavaScript. In the body of this function, the interface operation `op$bar` is called. The compiler prefixes the names of interface operations with `op$` to avoid naming clashes. Finally, the function `main` calls `work` and supplies the instance of `Foo` as an argument. This generated JavaScript code is logically equivalent to input program, written in `Effekt`.

```
const $effekt = require('./effekt.js')

var $examples_abstract = {};

var Foo = (function () {
    var module = {}

    var $Foo = {};

    function bar() {
        return $effekt.pure(40)
    }

    function baz() {
        return $effekt.pure(2)
    }

    return module.exports = Object.assign($Foo, {
        "bar": bar,
        "baz": baz,
        "op$bar": bar,
        "op$baz": baz
    })
})()

function work(mod) {
    return (mod.op$bar()).then((tmp73) =>
    (mod.op$baz()).then((tmp74) =>
    $effekt.pure($effekt.infixAdd(tmp73, tmp74))))
}

function main() {
    return (work(Foo)).then((tmp70) => $effekt.println(tmp70))
}

return module.exports = Object.assign($examples_abstract, {
    "main": main,
    "Foo": Foo,
    "work": work
})
```

Figure 5.3: JavaScript code generated from Figure 3.4 by the compiler

5.2 Symbols

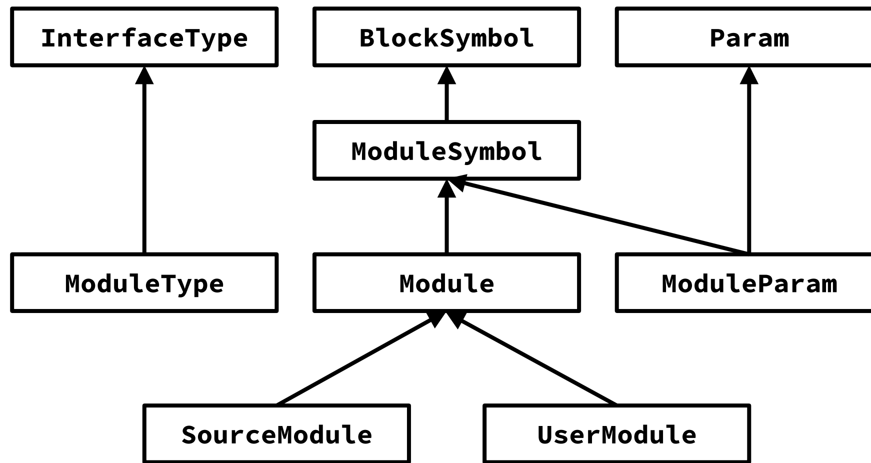

Figure 5.4: Simplified class-hierarchy of module-related symbols.

Compiler symbols represent instances of language entities. Usually each definition in the source code corresponds to the creation of a symbol, e.g. the line `module Hello { }` will be translated into a `UserModule` symbol with name `"Hello"` and a fresh id. As part of this thesis, the Effekt compiler was extended with additional symbols (Figure 5.4). A `ModuleType` (5.2.1) represents a module interface with a list of member operations. The trait `ModuleSymbol` (5.2.2) generalizes all second-class entities related to modules. This includes the `SourceModule` (5.2.3) and `UserModule` (5.2.4) symbols which represent sources and user-defined modules respectively. Finally, functions that take a module as parameter gain access to the members via a `ModuleParam` (5.2.5) symbol. The compiler uses those symbols to collect and distribute information about modules.

5.2.1 Module Type

There are similarities between the design of the `ModuleType` and `UserEffect` symbols. Both extend the `TypeSymbol` trait, have a user-defined `Name` and carry a list of `Method` symbols representing body-less function definitions. In fact, `Method` symbols are always owned by either a `ModuleType` or `UserEffect` symbol. This highlights the conceptual connection between interfaces and effect signatures. The `ModuleType` and `UserEffect` symbols mainly differ in two points: First, the `ModuleType` symbol also extends the `InterfaceType` trait. Second, effects can have type parameters which are not supported by

modules yet. While the second point might become obsolete when future versions of the compiler allow generic modules, the first difference presents the bigger hurdle for the unification of interfaces and effects. This is mainly due to the call-side handling of modules and effects. While members of a `ModuleType` are accessed through a user-defined module parameter, effect members are called on implicit passed capabilities managed by the compiler. Those distinctions require different handling of both symbols in various parts of the compiler. It is desirable to merge `ModuleType` and `UserEffect` symbols in the future to narrow the gap between modules and effects further.

5.2.2 Module Symbol

Every symbol which can be used as a module implements the `ModuleSymbol` trait. Therefore all symbols that can be passed as a module argument must implement that trait. Namely, this includes module parameters (`ModuleParam`), source modules (`SourceModule`) and user modules (`UserModule`). Thus all symbols that inherit from `ModuleSymbol` are second-class entities. Besides the usual requirement for symbols inside the `Effekt` compiler, a `ModuleSymbol` must provide a method `load():Scope` which (re)creates a scope, containing all member symbols of the module. Having a common trait shared by all kinds of modules helps to simplify the resolution of module members.

The abstract class `Module` implements shared logic for the `SourceModule` and `UserModule` symbols. Compared to `ModuleParam`, the classes of `SourceModule` and `UserModule` are more complex. Sources and user-defined modules can implement interfaces and export symbols from their scope, while a module parameter only provides access to the members of an interface. A `SourceModule` or `UserModule` exports symbols from a scope with the `save(scp:Scope)` method. Also, they need to store a list of implemented interfaces to provide the type-checking phase with sufficient information. Both aspects are covered by the `Module` base class, allowing the compiler to handle source and user modules indifferently.

All symbols representing actual modules are categorized as terms in the Effekt compiler. `ModuleSymbol` extends `BlockSymbol` which in turn is a sub-type of `TermSymbol`. Thus, module symbols life in a different namespace than types. Programmers can define, e.g. an user module and an interface with the same name without facing a name clash. The compiler will always be able to figure out if a given name references a `ModuleType` or `ModuleSymbol`. The `ModuleSymbol` trait integrates modules into the type-hierarchy of term symbols.

5.2.3 Source Module

The `SourceModule` symbol sits on top of the module hierarchy. The notation of a `SourceModule` existed prior this work on the compiler. Before modules were introduced into Effekt, the compiler already recognized source files as a single unit. However at this time, sources were neither first- nor second-class citizens of the language. They were mainly used to organize the compilation process and could only be used as imports. Many implementation details of the `SourceModule` class are carried over from previous versions. The main contribution to this symbols is their integration into the new module hierarchy, elevating them to second-class entities and allowing interface implementations.

The compiler represents every processed source with different instances of `SourceModule` symbols. Such a symbol contains all top-level declarations of a file. The name of a `SourceModule` is either explicitly declared using the `module` keyword in the first line of the file or implicitly derived from the file's name. A key feature of a `SourceModule` is the ability to import other sources. Those imports are represented as a list of `SourceModule` instances. Thus the relation between different sources forms a top-down tree, where the parent node holds a reference to its children. The `load()` method of source modules recursively iterates over all children. This results in a scope that contains all (transitively) imported symbols, but also accounts for shadowing.

5.2.4 User Module

The symbols of `UserModule` and `SourceModule` cover different use-cases for modules. Programmers can group multiple implementations of the same interface in one source with user modules. A `UserModule` comes with virtually no overhead and can access or shadow symbols from their parent scope. This makes `UserModule` a cheap alternative to `SourceModule`.

The `UserModule` is a child module with a parent of type `Module`. Since there are only two subclasses of `Module`, the parent hierarchy of a `UserModule` will eventually end with a `SourceModule`. This is due to the fact, that user-defined modules can always be associated with a single source containing their definition. The `load()` method of a `UserModule` walks up the parent chain. Thus, the scope is successively populated and respects the nesting of user modules. The parent structure of a `UserModule` distinguishes this symbol from other module symbols.

5.2.5 Module Parameter

Since modules in Effekt are second-class citizens, programmers must be able to pass them around as an argument. Thus, the compiler needs to represent module parameters in form of a `ModuleParam` symbol. Such a parameter is always linked to a `ModuleType`. Thus, the class `ModuleParam` implicitly integrates `ModuleType` symbols into the type-hierarchy of `ModuleSymbol`.

`ModuleParam` implements two important traits: First, it is a subtype of `Param`. This trait is used to model the parameter section of functions. Second, it is also a `ModuleSymbol`, allowing users to pass module parameters as arguments to subordinated function calls. The `load()` method of this symbol does not need recursive calls in order to restore the scope. It is sufficient to extract the `Method` symbols from the bound `ModuleType`. Thus a module parameter can be seen as a placeholder for an actual module.

Chapter 6

Conclusion

Modules and effects tackle common problems of complexity in software development. While modules provide a tool to structure and organize the components of an application, effects offer additional control over the program's flow. Both features complement each other and can coexist in one language. This finding is underlined by the integration of modules into the programming language Effekt. Furthermore, the concepts of effects and modules overlap in many instances. The definition of signatures, implementations and consumers of effects and modules follow a common pattern. However, they differ in their use-cases. Nevertheless, this raises the interest in possible combinations of modules and effects.

A main characteristic of Effekt's module system are second-class modules. This design decision is motivated by the implementation of effects in form of capabilities. Because those entities are also second-class, first-class modules would introduce additional barriers. However, the price of this choice is the exclusion of ML-style functors from the language. This eliminates a central construct for the composition of modules. Still, Effekt's module system can cover a variety of use-cases, including: abstraction, separate compilation and information hiding.

The current implementation of modules serves as starting point for further research. While the existing mechanism allow modules to provide effect handlers through interfaces, additional language features might desirable. This could include an uniform definition of module and effect signatures. Also, support for local module definitions can come handy. This would allow programmers to close a module implementation over capabilities, opening up new doors for possible research topics. Perhaps such a language feature could even compensate for the loss of functors. Finally, the potential of effects, modules and their combination is far from being exhausted.

Bibliography

[1] Rolf Adams, Walter Tichy, and Annette Weinert. "The cost of selective recompilation and environment processing". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 3.1 (1994), pp. 3–28.

[2] Nick Benton and Andrew Kennedy. "Exceptional syntax". In: *Journal of Functional Programming* 11.4 (2001), pp. 395–410.

[3] Dariusz Biernacki et al. "Binders by day, labels by night: effect instances via lexically scoped handlers". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–29.

[4] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. "Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428194. URL: https://doi.org/10.1145/3428194.

[5] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. "Semantic casts: Contracts and structural subtyping in a nominal world". In: *European Conference on Object-Oriented Programming*. Springer. 2004, pp. 365–389.

[6] Richard L Gauthier and Stephen D Ponto. "Designing systems programs". In: (1970).

[7] Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. "Continuations and coroutines". In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 1984, pp. 293–298.

[8] *ISO/IEC 14882:2020 - Programming language C++*. Standard ISO/IEC. Geneva, CH: International Organization for Standardization, Dec. 2020.

[9] Bill Joy et al. *The Java language specification*. 2000.

[10] Konstantin Läufer, Gerald Baumgartner, and Vincent F Russo. "Safe structural conformance for Java". In: *The Computer Journal* 43.6 (2000), pp. 469–481.

[11] Daan Leijen. "Koka: Programming with row polymorphic effect types". In: *arXiv preprint arXiv:1406.2061* (2014).

[12] Sam Lindley, Conor McBride, and Craig McLaughlin. *Do be do be do.* 2017. arXiv: 1611.09259 [cs.PL].

[13] David MacQueen. "Modules for standard ML". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.* 1984, pp. 198–207.

[14] Darya Melicher et al. "A capability-based module system for authority control". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[15] Ligia Nistor et al. "Wyvern: A simple, typed, and pure object-oriented language". In: *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and Inheritance.* 2013, pp. 9–16.

[16] Klaus Ostermann. "Nominal and Structural Subtyping in Component-Based Programming." In: *J. Object Technol.* 7.1 (2008), pp. 121–145.

[17] Leo Osvald et al. "Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect". In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 234–251.

[18] David L Parnas. "On the criteria to be used in decomposing systems into modules". In: *Pioneers and Their Contributions to Software Engineering.* Springer, 1972, pp. 479–498.

[19] Benjamin C Pierce. *Advanced topics in types and programming languages.* MIT press, 2005.

[20] Gordon Plotkin and John Power. "Algebraic operations and generic effects". In: *Applied categorical structures* 11.1 (2003), pp. 69–94.

[21] Gordon Plotkin and John Power. "Computational effects and operations: An overview". In: (2002).

[22] Gordon Plotkin and Matija Pretnar. "Handlers of algebraic effects". In: *European Symposium on Programming.* Springer. 2009, pp. 80–94.

[23] Gordon D Plotkin and Matija Pretnar. "Handling algebraic effects". In: *arXiv preprint arXiv:1312.1399* (2013).

[24] Christopher Strachey. "Fundamental Concepts in Programming Languages". In: *Higher Order Symbol. Comput.* 13.1–2 (Apr. 2000), pp. 11–49. ISSN: 1388-3690. DOI: 10.1023/A:1010000313106. URL: https://doi.org/10.1023/A:1010000313106.

[25] Allen Wirfs-Brock. “ECMAScript 2015 language specification”. In: *Ecma International,* (2015).

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Ort, Datum Unterschrift