Design and Implementation
of Effect Handlers
for Object-Oriented Programming Languages

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Jonathan Immanuel Brachthäuser, M.Sc.
aus Hanau

Tübingen
2019
Tag der mündlichen Qualifikation: 07.05.2020
Dekan: Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter: Prof. Dr. Klaus Ostermann – Universität Tübingen
2. Berichterstatter: Prof. Dr. Philipp Haller – KTH, Stockholm, Schweden
3. Berichterstatter: Prof. Dr. Peter Thiemann – Universität Freiburg
To my family, who always believed in me — and my wife for her love, patience, and unwavering support throughout the journey of life and thesis.
Acknowledgments

“A PhD is challenging but you are not alone – you will always have friends and colleagues.” — Paolo G. Giarrusso (2014)

My friend Paolo and I were sitting in the Sudhaus pub in Marburg, where this remark by Paolo finally convinced me to start doing a PhD. And, as it turned out, Paolo was right. In both regards. The times of my PhD studies were volatile and challenging. And, luckily, there were all these amazing people offering their friendship and support. I want to use this opportunity to express my gratitude to all of you who accompanied my journey – including many that are not mentioned in the following.

First of all, I want to thank my committee: Klaus Ostermann, Philipp Haller, Peter Thiemann, Torsten Grust, and Oliver Bringmann. In particular, I want to thank my supervisor Klaus Ostermann: Thank you for believing in me and giving me the opportunity to do the PhD with you. I am indebted to you for teaching me the theoretical foundations of programming languages. Writing my first academic paper with you for OOPSLA’14 was a great experience that eventually motivated me to do the PhD. I am also grateful for the freedom of choice to work on topics I personally find interesting. It was this freedom that kept me in the PhD program. In particular, thank you for enabling me to attend the POPL’17 conference in Paris. At that time I was not sure about my choice of topic and attending the conference gave me a completely fresh start. Despite that it also led to my luggage being stolen and a bad case of influenza.

At POPL’17, it was the talk by Daan Leijen on algebraic effects and his language Koka that motivated me to start working on algebraic effects. A year later, Daan gave me the opportunity to do a PhD research internship with him at Microsoft Research, Redmond. Thank you Daan! For your enthusiasm and positivity, for being a great person, a great collaborator, and a great mentor. Working with you is a delight and I hope we can continue our collaboration over the years to come.

I want to thank my Marburg friends and colleagues: Paolo Giarrusso, Tillmann Rendel, and Sebastian Erdweg. Discussions with every single one of you, showed me how interesting PL research is. From you I not only learnt a plethora of things about PL, but also the basics of working scientifically, how to write papers, and critical judgement. I am deeply indebted to all three of you, for being my mentors and friends at the same time. Thank you Paolo, for answering every single question on Stack Overflow! Thank you Tillmann, for teaching me how to tie my shoes the right way! Thank you Sebastian, for handing me down your Canne de Combat sticks!

Luckily, Paolo was right, and thus my PhD is the result of various collaborations. Every single collaboration was a pleasure and helped me to evolve, both personally and professionally. Thank you Tillmann, Paolo, Klaus, Matthias, Daan, Ningning, and especially Philipp for being awesome collaborators! Philipp, thank you for always challenging my ideas and for the fruitful and inspiring collaboration – I think we will never stop fighting about naming conventions. Thanks to everybody who gave feedback on early drafts of this thesis, including Matthias Hirzel, Ingo Skupin, David Binder, Philipp Schuster, and Tobias Stumpf.

I also want to thank my colleagues in Tübingen: Cai, Julia, Matthias, Philipp, Julian, Theo, David, Ingo, Luzia, and Stephen. I am very lucky to have had you as colleagues and friends in the last five years. Having you in the office, I was happy to go to work, every single day. Thank you for the many lunch-room discussions and random insights. The hardest part in working from home during the epidemic was to not have you around anymore. I also want to thank our
secretaries Birgit, Céline, and Bettina for keeping my back and fighting the uphill battle against bureaucracy.

I am grateful to my friends Michaël and Morgane for teaching me how to brew beer and for making Tübingen an interesting place to be!

Lastly, my eternal gratitude goes to my family and my wife. There are no words that can express how grateful I am to have you in my life.

Thank you all.
Abstract

An important aspect of software development is to structure the control flow of a program. Features to structure the control flow range from simple branching between two possible execution paths, to non-local control-flow transfers by means of jumps or exceptions. Furthermore, modern programming languages offer advanced control-flow structures like async/await to avoid blocking when performing input/output actions, generators to model demand-driven computation of streams, or coroutines to express control-flow transfer between different software components. Often, those advanced control-flow abstractions are built into the language and are thus not user definable. Users are thus limited to the control-flow features provided by the programming language designers and cannot create their own custom domain-specific control-flow abstractions.

Another aspect of software development is to parametrize a software component over details that might vary. Parametrizing a component with configuration, behavior, or other components, enables reuse of the same component in different contexts. Components can be configured statically, or dynamically at runtime. Manually parametrizing components can quickly become tedious, both for the component author who needs to design the component for parametrization, as well as the user of the component who needs to provide all parameters. This is especially the case for components that depend on other parametrized components. They often need to be parametrized by all transitive parameters of their dependencies.

Algebraic effects (Plotkin and Power, 2003) and their extension with handlers (Plotkin and Pretnar, 2009, 2013) offer interesting new ways to structure programs. Programs use effect operations, which, like normal functions, can receive arguments and return results. However, the implementation of effect operations is left open and depends on the context in which the program is executed. Effect handlers, much like exception handlers, handle effect operations and provide their implementation. Like throwing an exception, calling an effect operation transfers control to the corresponding handler. Unlike exceptions, the handler can resume the computation and thus transfer control back to the call site. Effect handlers support both aspects of software development concisely: they can express advanced control-flow structures as well as parametrization of software components. It has been shown in the literature that effect handlers can express many of the aforementioned control-flow structures as libraries, such as async/await, generators, and coroutines. At the same time, effect operations can conveniently be used to parametrize software components while handlers provide the configuration. Unifying both aspects also guarantees well-defined interaction between control flow and parametrization.

While algebraic effects recently gained popularity in the programming language research community, we identify two problems hindering adoption by a wider audience of programmers. Firstly, programmers are immediately confronted with the full generality of effect handlers. While effect handlers are expressive enough to model advanced control-flow structures, not all use cases require this expressivity. Secondly, algebraic effects and handlers have been conceived in the realm of functional programming languages. In consequence, most implementations of effect handlers are either standalone languages following the functional programming paradigm or are library embeddings into existing functional programming languages.

In this thesis, we propose solutions to the two aforementioned problems with the goal to facilitate adoption of effect handlers by a wider audience.

To address the first problem, we systematically present effect handlers as a combination of delimited control (that is, they allow control-flow transfers) and dynamic binding (that is, they allow parametrization). While the connection between effect handlers and delimited control has previously been established and it has been shown that effect handlers can express
dynamic binding, we discover that dynamic binding and effect handlers form a spectrum. We present a language design that embraces this spectrum. Increasing expressivity and conceptual complexity step-by-step, we introduce the three features ambient values, ambient functions, and ambient control. Ambient values coincide with dynamically bound variables and ambient control coincides with a slightly restricted form of effect handlers. The novel intermediate form of ambient functions enables abstraction similar to effect handlers, but without modifying the control flow: Ambient functions are dynamically bound, but statically evaluated. Introducing effect handlers from dynamic binding offers programmers an alternative way to approach handlers. They can incrementally learn and understand the different generalizations (ambient values, functions, and control). We hope this facilitates adoption of effect handlers by a wider audience.

To address the second problem, we present a library design, which embeds effect handlers in object-oriented programming languages. Our design embraces the object-oriented programming paradigm and we map abstractions of effect handlers to key abstractions of object-oriented programming. Combining the two paradigms not only enables programmers to use effect handlers in object-oriented programs, but also to use object-oriented programming abstractions to modularize effect handlers. Our design employs explicit capability-passing style. That is, instead of dynamically searching for a handler at runtime, we pass instances of handlers as additional arguments to methods. We present an implementation of our library design in the language Scala. It is based on a variation of a continuation monad for multi-prompt delimited control (Dybvig et al., 2007) to allow effect handlers to express advanced control-flow structures.

We study the extensibility properties gained by embedding effect handlers into an object-oriented programming language. While it opens up new dimensions of extensibility, the implementation still comes with two limitations. Firstly, being based on a monad user programs have to be written in monadic style. Secondly, the implementation is not effect safe and capabilities can escape the scope of their handler. This potentially results in runtime errors. We separately address the two limitations. Firstly, we present an implementation in Java that performs a type-selective continuation-passing style transformation by rewriting of bytecode. This way, user programs in the Java implementation can be written in direct-style overcoming the limitation of the Scala implementation. Secondly, we enhance our Scala library with an embedded effect system. We index the type of effectful programs with an intersection type to track the set of used effects. Expressing the set of used effects as an intersection type allows us to reuse Scala’s support for subtyping to express effect subtyping and to reuse Scala’s support for type polymorphism to express effect polymorphism.
Zusammenfassung


Handler im Rahmen der funktionalen Programmierung entwickelt worden. Infolgedessen sind die meisten Implementierungen dieses Sprachkonstrukts lediglich auf funktionale Programmierung ausgelagert. Dies gilt sowohl für eigenständige Sprachimplementierungen, als auch für eingebettete Implementierungen in Form von Programmmbibliotheken.


# Contents

1. Introduction .......................... 1
   1.1. Thesis Overview and Contributions .................................. 4
   1.2. List of Papers and Publications ............................................... 8
   1.3. Structure of the Thesis ............................................................ 13

I. Effect Handlers in Perspective  ....... 15
2. From Delimited Control to Effect Handlers .......... 17
   2.1. Delimiting Control ................................................................. 19
   2.2. Families of Delimited Control Operators ................................... 24
   2.3. From Delimited Control to Effect Handlers .................................. 27
   2.4. A Type System for Effect Handlers ............................................ 32
   2.5. Related Work and Chapter Conclusion ........................................ 36

3. From Dynamic Binding to Effect Handlers ........ 39
   3.1. Ambient Values ................................................................. 41
   3.2. Ambient Functions .............................................................. 45
   3.3. Ambient Control ................................................................. 49
   3.4. Type-Safety of Ambient Values ................................................ 54
   3.5. Ambient Values and Ambient Functions as Ambient Control ............. 59
   3.6. Translating to Effect Handlers and Back .................................... 63
   3.7. Related Work and Chapter Conclusion ........................................ 64

II. Effect Handlers and Object-Oriented Programming .... 67
4. Effekt – A Library Design .................. 69
   4.1. Programming with Effect Handlers in Effekt ................................ 72
   4.2. Delimiting Control ............................................................... 78
   4.3. Ambient State ................................................................. 83
   4.4. Composing Effect Signatures ..................................................... 84
   4.5. Composing Effect Handlers ....................................................... 87
   4.6. Composing Effectful Programs ................................................... 93
   4.7. Related Work and Chapter Conclusion ........................................ 96

5. Java Effekt – Effectful Programming in Direct Style .... 101
   5.1. Programming with Effect Handlers in JavaEffekt .......................... 103
   5.2. Implementing Effect Handlers for Java in three Steps ..................... 106
   5.3. Use Cases ................................................................. 112
# List of Figures

1. Effect handlers generalize exception handlers. They allow resuming the computation at the call site. Examples of an effect signature, an effectful program, and an effect handler in Scala-like pseudo syntax. ...................................................... 2
2. Overview of the different calculi presented in Chapters 2 and 3. The new feature of ambient values is highlighted. ................................................................. 5

2. Syntax and semantics of delimited control ($\lambda_{dc}$). Reduction rule ($shift$) is defined in-text. ............................................................ 21
3. Comparison of different control operators in the literature. ................................. 24
4. Generalization of syntax and semantics to families of control-operators ($\lambda_{dcp}$). .... 25
5. Syntax and semantics of multi-prompt delimited control with handlers ($\lambda_{dch}$). ... 29
6. Summary of syntax and semantics for $\lambda_{dch}$, specialized to variant $do+$ and a statically declared set of prompts (adapted from Leijen, 2017c). .......... 31

2. Syntax of types, typing rules, and equivalence of effect rows for $\lambda_{dch}$ (adapted from Leijen, 2017c). Every prompt $p$ has a uniquely defined signature in $\Sigma$. .... 33

3. Language of ambient values, $\lambda_{db}$ (adapted from Kiselyov et al., 2006). ........... 43
4. Extension of $\lambda_{db}$ with ambient functions ($\lambda_{dbf}$). ..................................... 47
5. Implementation of a depth-first traversal (adapted from Lewis et al., 2000). Call sites of ambient functions highlighted. ................................................... 49
6. Extension of $\lambda_{dbf}$ with ambient control ($\lambda_{dbc}$). ...................................... 53
7. Standard typing rules for $\lambda_{db}$ (adapted from Kiselyov et al., 2006, Figure 2). ...... 55
8. Effect safety for ambient values, functions, and control. Ambient rows $\pi$ are considered equivalent up to the order of the names in the row. .............. 57
9. Translating ambient values and ambient functions to ambient control. The translation is homomorphic except for the highlighted cases. ......................... 58
10. Translating the language of ambient control ($\lambda_{dbc}$) to delimited control with handlers ($\lambda_{dch}$) and back. Both translations are fully homomorphic. ...... 62

4. Mapping concepts from effect handlers to object-oriented programming. .......... 71
5. The control interface – a monad for multi-promp delimited control. .................... 73
6. Implementation of effect handlers for Exc and Amb using the library class Handler. 76
7. The prompt interface – control operators $shift$ / $reset$ and the marker trait $Prompt$. ................................................................. 78
8. Programming with effect handlers as structured programming with delimited control. .................................................................................. 80
9. Using answer-type-safe delimited control to declare and handle exception and ambiguity effects. ................................................................. 81
10. The State effect: mutable state that interacts well with multiple resumptions. .... 82
List of Figures

4.8. The handler implementation – handlers contain a prompt marker, use captures
the continuation and handle delimits the scope. ................................. 88
4.9. Handler for the Async effect – using two effects State and Fiber. .......... 90
4.10. Handler for the Fiber effect – using the State effect after capturing the continuation. 92
4.11. Representing stacks as prompt-separated list of frames. .................... 98

5.1. Structure of the JavaEffekt framework. Directed, solid arrows express dependencies.102
5.2. Example of using two effects in an effectful program. .......................... 103
5.3. Implementation of effect handlers for Exc and Amb using the library class Handler. 105
5.4. CPS translation of the example in Figure 5.4a, presented as a source-to-source transformation. .......................................................... 107
5.5. Interface and example implementation of the user-level stack. Implementations
of stack operations (push, pop and isEmpty) are left abstract. ................. 108
5.6. Implementation of control operators. Usage of the splittable stack implementation
Seq is highlighted. ................................................................. 109
5.7. The essence of the effect handler library: The Handler class. .............. 112
5.8. Effect signature Fiber with operations for cooperative multitasking and a round-
robin scheduler implemented as handler Scheduler. .............................. 116
5.9. Example of translating a method doLoop ....................................... 118
5.10. Type selective CPS translation via bytecode transformation. ................. 120
5.11. Performance of bytecode instrumentation libraries. Runtime in ms, lower is better.125
5.12. Performance of effect libraries. Runtime in ms, lower is better. ............ 127

6.1. The effect-safe control interface – changes compared to Figure 4.2 are highlighted.
Type alias Pure defined in text. ................................................. 134
6.2. The effect-safe prompt interface – changes compared to Figure 4.4 are highlighted.135
6.3. Using effect-safe delimited control to declare and handle exception and ambiguity
effects. Effect type related changes compared to Figure 4.6 highlighted. .... 138
6.4. Effect-safe state effect – effect signature of the built-in state effect and its handler
region. ............................................................................. 140
6.5. Implementation of effect handlers for Exc and Amb using the effect-safe library
class Handler – changes compared to Figure 4.3 are highlighted. .............. 141
6.6. Effect-safe versions of the Fiber effect signature and the Scheduler handler. All
term-level implementations are unchanged. ......................................... 143
6.7. Effect-safe versions of the Async effect signature and the Poll handler. All
term-level implementations are unchanged. ........................................ 145
6.8. Using Control to embed the λSC calculus into Scala. ......................... 154
Chapter 1

Introduction

Computer programming is intimately coupled to structuring control flow. From branching control flow based on conditionals, over employing a stack discipline when calling procedures, to using alternative return paths and transferring control flow with exceptions – thinking about control flow is a substantial aspect of every programmer's daily tasks.

The increasing need for distributed computing and non-blocking IO has led to the development of various advanced control-flow abstractions and programming models, such as asynchronous programming (that is, async/await) (Bierman et al., 2012), lightweight user-level threads and fibers (Dolan et al., 2013), and different forms of non-preemptive multitasking like asymmetric coroutines (Moura and Ierusalimschy, 2009) or generators (Politz et al., 2013). Each of these solutions is tailored to one particular programming problem. The way these abstractions are realized today is unsatisfactory. Often, they are either hard-coded into a language, (i.e., they are not user-definable), they are not composable (i.e., multiple abstractions cannot be used in one project), or implementations for one control-flow abstraction cannot be reused to implement similar other control-flow abstractions.

In contrast, algebraic effects (Plotkin and Power, 2003) in their extension with effect handlers (Plotkin and Pretnar, 2009) are a general control-flow structuring paradigm that is not tailored to a particular programming problem. Like Monads (Moggi, 1989), algebraic effects have originally been conceived as a framework to model the semantics of computational effects. Also like Monads, starting as a tool for semanticists, algebraic effects have then been rediscovered as a general program-structuring tool for programmers. However, there is a fundamental difference between the two concepts: Monads are centered on a type constructor, a semantic domain (like Either String a), from which effect operations (like throwError) emerge (Wadler, 1995). Algebraic effects instead start with the set of effect operations to then find a domain that supports the operations (Plotkin and Pretnar, 2009). Algebraic effects and handlers are thus operation centric, while monads are domain centric. Algebraic effects have their theoretical foundations in Lawvere theory (Plotkin and Power, 2003), but in this thesis we will view effect handlers more from the perspective of a software engineer. Following Kannmar et al. (2013), effect handlers can be thought of as a generalization of the simpler and widely known control-flow construct exceptions. Similar to exceptions, programs using effect handlers are conceptually split into three parts: Effect operations, effectful functions, and effect handlers. Figure 1.1 gives pseudo code examples for each of the three components. We use Scala-like pseudo syntax for easier comparison with the exception handling mechanism.

---

1. Introduction

Effect Signature: `effect Amb { def flip(): Boolean }

Effectful Program: `def choice(n: Int): Int = if (n > 1 && Amb.flip()) choice(n - 1) else n`

Effect Handler: `def collect[R](prog: () ⇒ R): List[R] = handle { Cons(prog(), Nil) } with Amb {
  def flip() = resume(true) ++ resume(false)
}

Figure 1.1. Effect handlers generalize exception handlers. They allow resuming the computation at the call site. Examples of an effect signature, an effectful program, and an effect handler in Scala-like pseudo syntax.

Effect operations

Compared with exceptions, effect operations correspond to `throw` as found in languages like Java (Gosling et al., 1996). However, instead of being built in, effect operations are user defined. One can define operations like `yield` to output elements of a (push-based) stream, `getHttp` to send (asynchronous) http-requests, or `suspend` to (cooperatively) transfer control to a scheduler (Leijen, 2016). Opposed to `throw`, effect operations can potentially return results. In the example in Figure 1.1, we follow Leijen (2017c) and declare the operation `flip` which returns a boolean.

Effectful programs

Programs are `effectful` (Kammar et al., 2013; Bauer and Pretnar, 2015) if they call effect operations. This can occur either directly or indirectly via other effectful programs. Like checked exceptions in Java, some implementations of effect handlers track the effects used by an effectful program in its type. This allows the type checker to distinguish effectful programs from pure programs and to assert that all effects are eventually handled. The effectful program `choice`, adapted from Danvy and Filinski (1992, p. 154), uses the `flip` operation to nondeterministically return a number between 1 and n.

Effect handlers

Effect handlers generalize exception handlers (Plotkin and Pretnar, 2009; Bauer and Pretnar, 2013). They implement the effect operations, specifying what it means for example to yield, send http-requests or suspend. Like exception handlers delimit the scope with `try { ... }`, effect handlers delimit the dynamic scope in which effect operations of a particular effect signature are handled by this very handler: The example handler, adapted from Plotkin and Pretnar (2013, p. 13), handles the operation `Amb.flip` for the extent of evaluating `Cons(prog(), Nil)`.

Effect operations are typically declared (and grouped) in effect signatures (such as `Amb`), which act as an interface between the user of effect operations (that is, effectful programs) and the implementer of effect operations (that is, effect handlers). Programming with effect handlers encourages modularity, by separating the declaration of effect operations from their implementation in effect handlers (Kammar et al., 2013).

While conceptually related, there is one important improvement of effect handlers over exception handlers: To handle effects and to implement effect operations, handlers get access to the delimited continuation, which is the remainder of the program execution – from the call of
the effect operation up to the enclosing effect handler (Plotkin and Pretnar, 2013). Provided
with the continuation, the effect handler can decide to resume the execution from the point
where the effect operation was originally called. This can also be understood from an operational
perspective. Following the analogy to exceptions, let us recall how traditional implementations
of exceptions perform a search for the corresponding handler. Exceptions unwind the stack
frame-by-frame, until a matching handler is found. The stack segment between the exception
throwing site and the handler is discarded (Gosling et al., 1996).

Like exception handlers, effect handlers can be thought of as marking positions on the runtime
stack. Also, the search for a corresponding handlers proceeds similar to exceptions. However, in
contrast to exceptions, effect handlers do not discard the unwound stack segment. Instead, they
reify the stack segment as a resumption function (Leijen, 2017c) (or continuation) and make it
available to the handler implementation (Plotkin and Pretnar, 2013).

Figure 1.1 defines the handler collect (Plotkin and Pretnar, 2013) which takes a program
prog potentially using the Amb.flip effect to compute a result of type \( R \). We assume a call-by-
value language and thus the program needs to be thunked to defer the effects (like flip) to
the point of handling. To handle the Amb.flip operation, the collect handler can access the
delimited continuation by using resume. Resuming with a value is similar to returning from
a normal method call. Computation will proceed at the call site of the effect operation flip.
However, like with exceptions, we can decide to never resume, or as in our example, we can
even resume multiple times. We resume once with true and once with false, enumerating
all possible outcomes of flip. Since the type of our handled program (Cons(prog(), Nil)) is
List[R], resuming will each time yield a list, which we finally concatenate to gather all results.

For example, calling
collect(() => (choice(2), choice(3)))
yields the result:
▶ List((1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3))

The ability to resume the computation at the call site of the effect operation is the impor-
tant ingredient that provides effect handlers with additional expressive power over exceptions.
Effect handlers can express the aforementioned control-flow abstractions as libraries (Dolan
et al., 2015, 2017; Wu et al., 2014; Kammar et al., 2013; Hillerström and Lindley, 2016; Lei-
jen, 2017a). Implementing abstractions like concurrent programming (Dolan et al., 2017) and
async/await (Leijen, 2017a) with effect handlers naturally allows a combined usage. This is good
news for both language implementers and programmers. Implementers who want to implement
multiple of the aforementioned control-flow structures now only need to support effect handlers.
Interaction between the features can be considered on the user-/ library-level. Furthermore, users
can use effect handlers to define custom (potentially domain specific) control abstractions.
1. Introduction

1.1 Thesis Overview and Contributions

Effect handlers are a powerful program-structuring paradigm. However, we identify two problems that potentially hinder a widespread adoption of effect handlers.

Firstly, as in the previous section, programmers are often immediately confronted with the full generality of effect handlers. In consequence, effect handlers are reported (Abramov, 2019) to have the reputation of being difficult to understand. While effect handlers are expressive enough to model advanced control-flow structures, not all use cases require the expressiveness to the full extent. For instance, to make the current request object available in a webserver implementation does not necessarily require the ability to capture the continuation.

Secondly, algebraic effects and handlers have been conceived in the realm of functional programming languages. In consequence, many implementations of effect handlers can be found in functional programming languages. They are either built into the language, like in Eff (Bauer and Pretnar, 2015), Koka (Leijen, 2014), Frank (Lindley et al., 2017), Multicore OCaml (Dolan et al., 2017), Links (Hillerström et al., 2017), and Helium (Biernacki et al., 2019). Or they are implemented as a libraries for Haskell (Kammar et al., 2013; Kiselyov et al., 2013; Wu and Schrijvers, 2015), OCaml (Kammar et al., 2013; Kiselyov and Sivaramakrishnan, 2016), or Idris (Brady, 2013). With the notable exception of a recently presented object-oriented calculus JEff (Inostroza and van der Storm, 2018), effect handlers are not widespread in the realm of object-oriented programming (OOP).

In this thesis, we propose solutions to the two aforementioned problems with the goal to facilitate adoption of effect handlers by programmers. The thesis is structured in two parts, reflecting the two different ways in which we work towards this goal.

In the first part, we systematically present effect handlers as a combination of delimited control and dynamic binding. We follow Moreau (1998) and refer to dynamically scoped variables as dynamic binding. We use late binding (Suzuki, 1981) to refer to dynamic binding of method implementations in OOP. Step-wise generalizing dynamic binding to effect handlers, we discover that dynamic binding and effect handlers form a spectrum and present a language design that embraces this spectrum. Introducing effect handlers from dynamic binding offers programmers an alternative way to approach handlers. They can incrementally learn and understand the different generalizations.

In the second part, we present a library design that embeds effect handlers in object-oriented programming languages. Our design embraces the OOP paradigm and we map abstractions of effect handlers to key abstractions of OOP. Combining the two paradigms not only enables programmers to use effect handlers in object-oriented programs, but also to use OOP abstractions to modularize effect handlers.

We now give a more detailed overview over the two parts and summarize our contributions.

1.1.1 Effect Handlers in Perspective

In the first part of this thesis, we establish the necessary terminology and present effect handlers as a combination of two seemingly orthogonal features: delimited control and dynamic binding. The developments of this part of the thesis are illustrated in Figure 1.2. To introduce terminology and to discuss the abstractions provided by effect handlers, we summarize existing work in different calculi (Figure 1.2). In Chapter 2, we follow historical developments in the literature on control effects and present effect handlers in the light of delimited control (Felleisen, 1988). In Chapter 3, we then perform similar steps of generalization, but starting from dynamic binding (Moreau, 1998).
Both features, delimited control and dynamic binding, have a somewhat bad reputation and naively combining them as two individual features comes with problems (Kiselyov et al., 2006). Nevertheless, as a type- and effect-safe combination of both delimited control and dynamic binding, effect handlers occupy a sweet spot in the design space between the two features. Both, the relation between delimited control and effect handlers (Forster et al., 2017), as well as the one between dynamic binding and effect handlers (Kammar and Pretnar, 2017) have been established previously. However, we believe that introducing effect handlers from dynamic binding offers programmers an alternative way to approach programming with effect handlers. Increasing expressiveness and conceptual complexity step-by-step, we introduce the three features ambient values, ambient functions, and ambient control. Each step highlights a different, important aspect of programming with effect handlers. Importantly, our systematic presentation helps us to discover the novel feature of ambient functions, which resides between dynamic binding and effect handlers. Ambient functions enable abstraction similar to effect handlers, but without modifying the control flow: They are dynamically bound, but statically evaluated.

Related publications The first part of this thesis is based on the following articles:


Contributions The first part of this thesis makes the following contributions:

- We systematically present two aspects of effect handlers. Once starting from (delimited) control operators (Chapter 2), and once starting from dynamic binding (Chapter 3). The two developments lead to two equivalent calculi ($\lambda_{dch}$ and $\lambda_{dbc}$).

- While it has been previously established that effect handlers can express dynamic binding (Kammar and Pretnar, 2017), we observe that dynamic binding and effect handlers form a spectrum. We perform a step-wise generalization from ambient values (dynamic
1. Introduction

binding) to ambient control (effect handlers). This leads us to a novel intermediate form of abstraction: ambient functions (Section 3.2).

– We present a language design that incorporates all three generalization steps (ambient values, functions, and control). We use the macro translation from ambients to effect handlers (Sections 3.5 and 3.6) to implement the language design as an extension of the programming language Koka (Leijen, 2017c).

– We present a type- and effect-system for ambient values, functions, and control (Section 3.4.2). Expressing ambient values and functions in terms of ambient control (Section 3.5) allows us to reuse existing soundness proofs (Sections 2.4 and 3.6).

1.1.2 Effect Handlers and Object-Oriented Programming

The second part of this thesis represents our main contribution. We present Effekt – a library design for embedding effect handlers in object-oriented languages. There are two important aspects of our library design.

Firstly, as opposed to many other effect libraries and language implementations, Effekt is not based on a free monad representation (Kiselyov and Ishii, 2015) but instead centers around the concept of explicit capability-passing style on top of a runtime system for multi-prompt delimited control (Hieb and Dybvig, 1990; Dybvig et al., 2007). That is, instead of searching for the correct handler implementation of an effect operation at runtime, we explicitly pass capabilities as additional arguments to effectful functions. Capability passing helps us to understand effects as imposing an additional requirement on the calling context, rather than a side effect that occurs in addition to computing the result. As we will see in Chapter 6, effects are thus contravariant.

Secondly, programming with effect handlers in Effekt is object-oriented programming. Our library design of Effekt is designed to integrate well with object-oriented programming languages and we directly map abstractions of effect handlers to abstractions in object-oriented programming. Effect signatures are interfaces declaring effect operations, effect handlers are classes implementing those interfaces and capabilities are instances of effect handlers. Effectful programs are written against the abstraction of effect signatures. Combining the paradigms not only enables programmers to use effect handlers in object-oriented programs, but also to use object-oriented programming abstractions to modularize effect handlers.

We evaluate our design in two implementations. A library implementation in Java (Gosling et al., 1996), which we call javaEffekt (Section 5) and a library implementation in the programming language Scala, which we call ScalaEffekt (Sections 4 and 6). The two implementations both perform capability passing and rely on an implementation of multi-prompt delimited control. While the programming interfaces are similar, their implementations are quite different.

Our Java implementation consists of three core components: (1) A type selective continuation-passing style (CPS) transformation (Fischer, 1972; Reynolds, 1972) via JVM bytecode transformation, (2) an implementation of delimited continuations (Felleisen, 1988) on top of the bytecode transformation, and finally (3) a library for effect handlers in terms of delimited continuations. The CPS transformation allows us to implement the resumption behavior of effect handlers, while user programs can be written in direct style.

In contrast, our Scala implementation is based on a monadic implementation of delimited control (Dybvig et al., 2007). Consequently, users have to write effectful code in monadic style. Another important difference is that ScalaEffekt (as presented in Chapter 6) is effect-safe, while in javaEffekt capabilities can leak, which might result in runtime errors.
1.1. Thesis Overview and Contributions

Related publications The second part of the thesis is based on the following articles:


Contributions The second part of this thesis makes the following contributions:

- We present the first library design of a type-safe embedding of effect handlers that revolves around object-oriented programming. The design is based on capability-passing style as an alternative to dynamic binding of effect handlers. In our library design, programming with effect handlers is object-oriented programming. Effect signatures correspond to interfaces; handlers correspond to implementations of the interfaces (Chapter 4).

- We implement the design in two libraries, one for Java (Chapter 5) and one for Scala (Chapters 4 and 6).

- For our implementation of JavaEffekt, we present an implementation of multi-prompt delimited continuations in Java. It uses trampolining and avoids the typical linear overhead of restoring the stack upon resumption common to all continuation libraries in Java that we are aware of (Section 5.2.2).

- We present a type-selective, signature preserving CPS transformation of JVM bytecode. We use closures introduced in Java 1.8 to create specialized instances of continuation frames. The general idea is applicable to any VM-based language that supports closure creation (Section 5.4).

- For our implementation of ScalaEffekt, we build on the operational semantics of Dybvik et al. (2007) but make it effect-safe (Section 6.1). We achieve effect safety by generalizing techniques of Launchbury and Sabry (1997) to nested regions (Kiselyov and Shan, 2008) but using intersection types of abstract type members instead of rank-2 types (Chapter 6).

- We study the extensibility gained by combining effect handlers with object orientation; in particular Scala’s traits and mixin composition. We identify the effect expression problem as an instance of Wadler’s (1998) expression problem and show how ScalaEffekt supports the required dimensions of extensibility (Section 4.4).

- We evaluate the performance of our Scala and Java implementations of Effekt (Section 5.6.2). We compare the performance of programs translated with our bytecode instrumentation with other Java bytecode instrumentations (Section 5.6).
1. Introduction

1.2 List of Papers and Publications

The work on this thesis resulted in a number of paper submissions and publications at international conferences and workshops. Here, we briefly summarize the main contributions of each paper and discuss its relationship to the other papers and this thesis.

1.2.1 Effekt: Effect Handlers and Object-Oriented Programming

In the following articles, we presented different aspects of our library design.

Effect Handlers as a Monadic Library in Scala


In this paper, we introduce a Scala library for programming with effect handlers. The library design is based on the concept of explicit capability-passing style. Viewing effect operations as constructors of an embedded domain specific language (Hudak, 1998), capability-passing style corresponds to a shallow embedding (Carette et al., 2007; Hofer et al., 2008). Taking this point of view, we describe the effect expression problem as a variant of Wadler’s “Expression Problem” (Wadler, 1998). It is the first effect handler library to make extensive use of Scala’s new feature of implicit function types (Odersky et al., 2017). Implicit function types allow us to reduce most of the syntactic overhead associated with capability passing. To support continuation capture, we implement our library on top of a monad for multi-prompt delimited control (Dybvig et al., 2007). Preliminary benchmarks suggest that capability passing offers significant performance improvements over an established Scala library (Torreborre, 2016), which is based on freer monads (Kiselyov and Ishii, 2015).


While passing first-class capabilities offers an interesting alternative to implement effect handlers, it does not guarantee effect safety. Effect handlers introduce capabilities, which are only valid in the dynamic scope of the effect handler. Since capabilities are first class, they can leave the scope of the handlers, which potentially results in a runtime exception. In this paper, we add an effect system to our library implementation to ensure that capabilities can only be used in the dynamic scope of the corresponding handler. The effect system is inspired by the one of Zhang and Myers (2019). In our Scala implementation of the embedded effect system, we use intersection types and path-dependent types to guarantee well scopedness of capabilities. Trying to run a program that uses a capability outside of its corresponding handler results in a Scala type error. Using the existing Scala type-system to express effect safety of our library
embedding has the advantage that we can reuse Scala’s support for subtyping to express effect subtyping and Scala’s support for polymorphism to express effect polymorphism.

Effect Handlers via Bytecode Transformation in Java


In this paper, we evolve the original library design of ScalaEffekt to integrate better with Java. The efforts to integrate with Java result in a design, which directly maps effect signatures to interfaces, handlers to implementations, and capabilities to handler instances. While in ScalaEffekt, effectful user programs have to be written in monadic style, users of JavaEffekt write effectful programs in direct style. That is, programmers can use existing control-flow structures such as conditional branching and loops to describe effectful programs. To enable this, in the paper we introduce a type-selective, signature preserving CPS transformation of JVM bytecode. An implementation of multi-prompt delimited continuations on top of the instrumented bytecode allows capturing the continuation. The implementation performs trampolining and avoids the typical linear overhead associated with restoring the stack upon resumption. All continuation libraries in Java that we are aware of suffer from this linear overhead.

Effekt in this Thesis

The introduction of Chapter 2 is partially based on the second article (Brachthäuser et al., 2020). Chapter 4 is based on the three articles (Brachthäuser and Schuster, 2017; Brachthäuser et al., 2018, 2020) and presents Effekt, a library design centered on capability passing. The library is designed to blend into object-oriented languages with interfaces and classes. Section 4.6.1 briefly discusses the use of implicits to remove the burden of passing capabilities (Brachthäuser and Schuster, 2017). Otherwise, in this thesis we mostly refrain from using the feature of implicit function types to make capability passing more explicit. Section 4.4 introduces the effect expression problem (Brachthäuser and Schuster, 2017) and describes different dimensions of extensibility, enabled by the design of Effekt.

Chapter 5 is closely based on the third mentioned publication (Brachthäuser et al., 2018) and presents the implementation of JavaEffekt and our CPS transformation. Compared to the publication, this thesis contains additional benchmark results. The results illustrate the linear overhead of other continuation libraries and provide further evidence for the difference in asymptotic complexity (Section 5.6). Chapter 6 is closely based on the second mentioned article (Brachthäuser et al., 2020) and enhances the library with an embedded effect system. For a uniform presentation, and to facilitate comparison between the different stages of development, the library interfaces presented in Chapters 4 to 6 slightly differ from the original presentations in the corresponding articles.
1. Introduction

1.2.2 From Dynamic Binding to Effect Handlers


It has been previously established that effect handlers can express dynamic binding (Kammar and Pretnar, 2017), that is, dynamically scoped variables.

In this paper, we take the opposite approach and start from dynamic binding. We step-wise generalize dynamic binding from implicit values to implicit control. While implicit values correspond to dynamic binding (Lewis et al., 2000; Kiselyov et al., 2006), implicit control closely corresponds to effect handlers (Plotkin and Pretnar, 2013).

We discover that dynamic binding and effect handlers form a spectrum and propose the novel feature of implicit functions, which (from an expressivity point-of-view) resides between the two other features. Like implicit values, implicit functions are bound dynamically. Like implicit control (and effect handlers), the body of implicit functions is evaluated in the lexical scope of its definition. We show how the small generalization from regular implicit values to implicit functions allows encapsulating control-effects at the definition site of the function, as opposed to leaking them (in the type and operationally) at the call site of the function.

We formalize the new feature as an extension to Moreau’s calculus of dynamic binding (Moreau, 1998) and show how all three features are macro-expressible (Felleisen, 1990) in terms of effect handlers. Both, the design of a language with implicit functions, values, and control as well as the implementation on top of the Koka language (Leijen, 2017c) are the result of a PhD research internship at Microsoft Research, Redmond, USA. The formalization of the type- and effect systems in the article (Brachthäuser and Leijen, 2019) and the corresponding soundness proofs are largely due to Daan Leijen.

Ambient Functions in this Thesis

Chapter 3 is closely based on this paper. To match the most recent implementation in the Koka language (Leijen, 2017c), in Chapter 3 we renamed implicit values, functions, and control to ambient values, ambient functions, and ambient control, correspondingly. In the paper, we translate implicit (i.e., ambient) values, functions and control to effect handlers. Instead, in Section 3.5 of this thesis we show how ambient values and functions can be macro expressed in terms of ambient control. We adjust the corresponding proofs of type- and semantics preservation. In this thesis, ambient control coincides exactly (syntactic differences notwithstanding) with the simplified effect handlers of Section 2.4. The type- and semantics preserving translation of ambient functions and values to ambient control (and effect handlers in extension) allows us to reuse the proof of semantics soundness for effect handlers (Section 2.4.1 which is adapted from Leijen (2017c)).
1.2. List of Papers and Publications

1.2.3 Software Modularity


In this paper, we analyze the relationship between object algebras (Oliveira and Cook, 2012) and attribute grammars (Knuth, 1968).

Many application domains require traversing tree-structured data. Amongst the most prominent examples are interpreters and compilers that traverse the abstract syntax tree to evaluate or translate the source program. There exists a multitude of techniques, based on the idea of separating the recursive traversal from the computation at each node. Examples are the visitor pattern (Gamma et al., 1995), folds (or “catamorphisms”) (Meijer et al., 1991), shallow embeddings (Carette et al., 2007), and Church encodings (Barendregt, 1984, 1992). Object algebras (Oliveira and Cook, 2012; Oliveira et al., 2013) are a recent approach to structure tree traversals, similar in spirit to the aforementioned techniques. Programs written with object algebras can be structured in a pleasingly modular and extensible way. At their foundation, object algebras correspond to folds and the computation at a given node is expressed compositionally in terms of the results of the child nodes. This limits the class of programs that can be expressed in terms of object algebras.

In contrast, attribute grammars (Knuth, 1968) allow declarative specification of attributes as part of context-free grammars. Attributes on one node can be described in terms of attributes on other nodes, such as the parent node, child nodes, or sibling nodes. Attributes that can be computed bottom-up are called synthesized; attributes that are computed top-down are called inherited. Grammars with only synthesized attributes are called S-attributed (Knuth, 1968; Lewis et al., 1974).

In the paper, we show that object algebras as presented by Oliveira and Cook (2012) can be seen as a Church encoding of S-attributed grammars. We extend the expressiveness of object algebras in order to support traversals that correspond to richer classes of attribute grammars. We allow to additionally express dependencies (1) on attributes computed for child nodes, (2) on attributes computed for the current node, and (3) attributes computed on left siblings of the current node. With the latter, we are able to encode the full class of L-attributed grammars – a class that can express one-pass compilers. Implementing L-attributed grammars with object algebras has several advantages. Attributes can be defined and type-checked separately while dependencies between attributes are statically checked.

It is our prior work on this paper, which sparked the idea for shallow embeddings of effect handlers (Chapter 4). Effect handlers are often introduced as a fold over a computation tree (Kammar et al., 2013; Lindley, 2014). If handlers are folds, can we express them as a shallow embedding to inherit the associated modularity and extensibility benefits? The idea of shallowly embedding effect handlers resulted in explicit capability-passing style, which is an important aspect of the Effekt design. In Section 4.4, we study the extensibility properties that Effekt inherits from shallow embeddings.
1. Introduction

Programs that traverse tree-shaped data often suffer from the expression problem (Wadler, 1998), an important and well-studied problem in software engineering. The expression problem describes the difficulties of extending a recursive data type with new variants, while at the same time being able to modularly add new functions over the data type. With the notable exception of Lämmel and Rypacek (2008), the expression problem is often described only informally in the literature. Following Lämmel and Rypacek, in the paper, we formally model the expression problem using category theory. We then introduce the co-expression problem by applying categorical duality. While the expression problem centers on extending the least fixed point of a sum type and recursive functions consuming it — the co-expression problem is concerned with extending the greatest fixed point of a product type and co-recursive functions producing it.

In our papers on Effekt (Brachthäuser et al. 2017, 2018, 2020) and in Section 4.5.2 of this thesis, we use the intuition gained through the work on the (co-)expression problem to formulate the effect expression problem — a variant of the expression problem in the context of effect handlers.

Brzozowski derivatives (Brzozowski, 1964) are a well-known concept in the context of regular expressions. Recently, they have been rediscovered to give a simplified explanation to parsers of context-free languages (Danielsson, 2010; Might et al., 2011; Adams et al., 2016). Might et al. use the derivative of a parser as part of the implementation. In this paper, we add derivatives as a novel first-class feature to the interface of a standard parser combinator language. First-class derivatives enable an inversion of the control flow. This allows modularly implementing parsers for languages that previously required separate pre-processing steps or crosscutting modifications of the involved parsers. This opens up new opportunities for reuse and supports a modular and declarative specification of layout-sensitive parsers.

While at first parsing and effect handlers might seem like two very different fields, there are interesting connections between the two. Firstly, it has been shown that effect handlers can be used to implement parsers (Leijen, 2016). In Section 4.6, we will briefly revisit this approach and see how grammars can be expressed in terms of effect operations, and how the semantics of parsers is implemented as handlers for the effect operations. In general, by varying the effect handlers, we can freely choose between different parsing strategies: Depth-first or breadth-first parsing, computing only the first result or enumerating all results. Secondly, as observed by Kiselyov (2007), delimited control (and effect handlers in extension) can be used to invert the control flow and to incrementally provide the parser with input. This way we can implement online (that is, streaming) parsers (Swierstra, 2009), or use the results of our paper to add first-class derivatives to a parser combinator language.
1.3 Structure of the Thesis

1.2.4 Efficient Compilation of Effect Handlers


Despite existing optimization efforts (Pretnar et al., 2017; Leijen, 2017b; Dolan et al., 2015), to the current day, the abstraction of effect handlers still comes with a significant runtime cost that is associated with searching the correct handlers and capturing the continuation. A well-known translation strategy for delimited control in general (Danvy and Filinski, 1990), and for effect handlers in particular (Hillerström et al., 2017; Leijen, 2017c) is to compile to continuation-passing style. As a first step towards efficient compilation of effect handlers, in this paper, we combine the results of Danvy and Filinski (1990) and Danvy and Filinski (1992) and present a compilation strategy for multiple level of delimited control. We implement the translation of terms and of types in Idris. Our translation preserves typability by intrinsically typing both source and target language. We additionally index the type of effectful expressions by a list of answer types. This indexing serves two purposes: (1) it restricts the contexts in which this expression can be used and guarantees effect safety, and (2) it drives our type-directed compilation process. Each entry in the list of answer types results in one iteration of the CPS translation. We generalize standard techniques (Danvy and Filinski, 1992) to the iterated translation, in order to avoid multiple classes of administrative beta and eta-redexes.


Building on the results of this thesis, in the paper we generalize the techniques of the previous paper from delimited control to effect handlers. In particular, like in Effekt, we employ explicit capability-passing style. We present a calculus that guarantees full elimination of redexes associated with handler abstractions. To achieve this, we restrict capabilities to be second-class, which prevents them from being returned from functions. Passing capabilities at compile-time allows us to always specialize effectful programs to the corresponding handler implementations. While the inlining guarantees are powerful, our proposed solution comes with restrictions. Handlers are second-class, which implies that programmers cannot abstract over handlers at runtime.

1.3 Structure of the Thesis

The remainder of this thesis is structured according to the aforementioned two parts. The first part provides the necessary overview over delimited control and dynamic binding and relates both to effect handlers. The second part describes the library design of Effekt and discusses new modularization opportunities enabled by combing effect handlers with object-oriented programming. Chapter 7 discusses our approach, sketches potential future work and concludes.
Part I.

Effect Handlers in Perspective
Chapter 2

From Delimited Control to Effect Handlers

In this chapter, we follow historical developments in the literature and first generalize control effects to delimited control to then generalize from one control operator to a family of control operators. We finally introduce effect handlers by shifting the perspective on multi-prompt delimited continuations (Hieb et al., 1994; Dybvig et al., 2007): With effect handlers the implementation of an effect operation is localized at the handler, not at the effect operation (Sitaram, 1993).

Neither the overview, nor the generalizations bear much technical novelty and have been given in similar form by Shan (2004b) and Dybvig et al. (2007). Furthermore, the relation between effect handlers and control operators has been studied formally by Forster et al. (2017). However, presenting the different control operators in the context of effect handlers helps us to discuss modularity properties of the various control operators and effect handlers. Furthermore, this background chapter serves two purposes: firstly, it establishes the necessary vocabulary and concepts. The different calculi presented in this chapter serve as a point of reference for the rest of this thesis. In particular, we use the variant of multi-prompt delimited control to implement our effect handler libraries in the second part of this thesis. Secondly, it highlights that programming with effect handlers can be seen as structured programming with delimited control (Kammar et al., 2013, pp.150-151). This point of view is the foundation for explicit capability-passing style, established in later chapters.

Cartwright and Felleisen describe an effect as

“a closed expression that [...] is not a value, and cannot be reduced to a value without affecting the context.” — Cartwright and Felleisen (1994, p. 248)

As opposed to pure expressions, which can be reduced to a value without regarding the context (assuming well-typed expressions and neglecting non-termination), the evaluation of effectful expressions might depend on or even modify the context. While in general, the context can include resources like the heap, the file system, or I/O, in this work we focus on the syntactic evaluation context (Wright and Felleisen, 1994). We refer to effects that affect the evaluation context also more specifically as control effects and to built-in or user-defined operations that have control effects as control operators (Felleisen and Friedman, 1986).


2. From Delimited Control to Effect Handlers

For multiple decades, control operators like `call/cc` have been used to program with control effects (Friedman et al., 1984). Similar to `goto`, which can be understood as undelimited, local continuation (Landin, 1965; Kennedy, 2007), `call/cc` captures the undelimited global continuation, i.e., the entire future of the program execution. We will shortly see an example, but abstractly, invoking `call/cc`, captures the continuation, reifies it as a first-class value, and makes it available to the programmer. When the continuation is called, execution proceeds at the very point where `call/cc` was originally called (Haynes and Friedman, 1987). Such a captured continuation is undelimited, since it spans the entire future, not just some parts of it. It is global in the sense that it potentially spans across procedure boundaries.

Recently, in disguise, control effects have found their way into mainstream programming languages in the form of specialized solutions such as async/await, fibers, coroutines, generators and others. While many of these programming language constructs can be implemented in terms of other control effects (Wand, 1980; Haynes et al., 1986), programming language designers often refrained from directly offering control operators like `call/cc` to the users. And rightfully so! Like `goto`, while being very general and expressive, programs written with the control operator `call/cc` tend to be fragile, hard to understand, and difficult to maintain. In contrast, each of the programming language constructs listed above focuses on one particular problem domain. Each offers domain terminology, potentially specialized typing rules and semantics, seemingly reducing the complexity for the user of the programming language. However, the specialization to one particular solution comes with a price: Each feature is typically implemented individually in the compiler or runtime. Even worse, combining multiple features in one programming language often leads to interaction of the features, again increasing the complexity of the language for both the user and the implementer.

Also recently, the programming languages research community found new interest in control effects in the form of algebraic effects (Plotkin and Power, 2003) and their extension with handlers (Plotkin and Pretnar, 2009, 2013). Effect handlers occupy a sweet spot between the very general control operators (such as `call/cc`) and specialized programming language features (such as async/await). Like general control operators, effect handlers are powerful and can express many of the language features above as user-defined libraries (Dolan et al., 2015, 2017; Leijen, 2017a). However, unlike general control operators, effect handlers also encourage modularity by separating programs into effect signatures (interfaces listing the available effect operations), effectful programs written against those signatures, and effect handlers implementing the effect signatures thereby providing semantics to the effect operations (Kammar et al., 2013). This separation improves modular reasoning, simplifies type checking, and enables well-defined composition of programs using different effects.

The connection between effect handlers and delimited control is not accidental. It has been established practically (Kiselyov and Sivaramakrishnan, 2016, 2018) as well as theoretically (Forster et al., 2017; Piróg et al., 2019) that certain forms of delimited continuations can express certain forms of algebraic effect handlers. In the literature, effect handlers are sometimes introduced as a structured way to programming with delimited continuations (Kammar et al., 2013; Leijen, 2017c).

"Effect handlers are to delimited continuations as structured programming is to goto." — Andrej Bauer (Dagstuhl Seminar 18172, March 2018)

Viewing programming with effect handlers as structured way of programming with delimited continuations has merits on its own.
In particular, we believe the regained interest in control operators comes from four important generalizations and improvements effect handlers make over call/cc:

1. Generalizing from undelimited to delimited continuations
2. Generalizing from one control operator to a family of control operators
3. Establishing answer type safety of control operators
4. Establishing effect safety of control operators

From an engineer’s perspective, each of these improvements helps to write programs in a modular way, making them easier to extend, and making it easier to reason about parts of a program in isolation. While effect handlers can be seen as a revival of structured programming with delimited control operators, there exists an alternative point of view we want to emphasize: effect handlers generalize dynamic binding (Chapter 3). Hence, effect handlers encapsulate two powerful (but often frowned upon) features behind a single uniform abstraction.

**Delimited control**

Effect handlers can capture the (delimited) continuation when handling effect operations – effect operations are thus related to control operators and effect handlers are related to control delimiters (Kammar et al., 2013).

**Dynamic binding**

Effect operations are dynamically bound (or dynamically scoped (Moreau, 1998)) – effect operations are thus related to dynamically bound (effectful) functions and effect handlers are related to binders of such dynamically scoped (Kammar and Pretnar, 2017).

In this chapter, we follow the above generalizations in order to establish important terminology and to better understand the relation between effect handlers and delimited control. We review the design space of delimited control, explore different variants of delimited control operators, generalize from one control operator to families of control operators, to finally identify one particular variant (spawn / controller) that most closely matches the operational semantics of effect handlers in our implementations of the second part of this thesis.

### 2.1 Delimiting Control

The control operator call/cc (as present in some Scheme dialects) captures the entire continuation of the program.

**Example**

```plaintext
print((1 + callcc \{ k ⇒ k(10); k(20) \}) * 2)
```

In this example, reducing callcc captures the entire rest of the program execution as continuation \( k \). The continuation corresponds to the evaluation context \( E = \text{print}((1 + □) * 2) \). Invoking the continuation with \( k(10) \) aborts the computation and replaces the current evaluation context with \( E \). The example thus evaluates to \( E[10] \) and running it prints the number 22. We can also understand callcc as a labeling instruction and calling the continuation \( k(10) \) as a jump to that label with the given argument. This analogy to jumps also makes it immediate that calling the continuation discards the current continuation (that is, the second call \( k(20) \)) and replaces it with \( \text{print}((1 + 10) * 2) \). The call to \( k(10) \) never returns.
2. From Delimited Control to Effect Handlers

A first step towards effect handlers is to allow users to delimit the extent to which the continuation is being captured. In variations, such a delimited continuation (Felleisen, 1988) is also referred to in literature as subcontinuation (Hieb et al., 1994) or partial continuation (Johnson and Duggan, 1988). Since their introduction by Felleisen, different variants of delimited control operators have been introduced – maybe the most prominent example being shift / reset (Danvy and Filinski, 1992). Using shift, we can write a variation of the above example as:

\[
\text{print}(\text{reset}\{ (1 + \text{shift}\{ k \Rightarrow (k(3) + k(4)) \}) \ast 2 \})
\]

Now the continuation (highlighted in grey) corresponds to the evaluation context \((1 + \square) \ast 2\). Its extent is delimited by reset and thus does not include the call to print. The example will print the number \(((1 + 3) \ast 2) + ((1 + 4) \ast 2) = 18\). Importantly, in contrast to the example with call/cc, the call to the delimited continuation \(k(3)\) does return with the value 8.

2.1.1 Four Variants of Delimited Control

Figure 2.1 formally captures the operational semantics of delimited control. It defines the syntax and semantics of a calculus for a call-by-value language with delimited control \(\lambda_{dc}\). The style we present the calculi in this and the following chapter is adapted from Leijen (2017c), who in turn uses the style of Wright and Felleisen (1994). The syntax includes the usual forms for lambda abstraction \((\lambda x. e)\) and application \((e_1(e_2))\), variables \((x)\) and constants \((c)\). In addition, it also includes the two special forms for capturing the continuation \((\text{shift})\) and delimiting the extent of the captured continuation \((\text{reset})\). In addition to the standard abbreviations for curried function definition and application, we use the following abbreviations:

\[
\begin{align*}
\text{val } x &= e_1; e_2 \equiv (\lambda x. e_2)(e_1) \\
\text{def } f(x) &= e_1; e_2 \equiv \text{val } f = \lambda x. e_1; e_2 \\
& \quad = \text{val } _{} = e_1; e_2
\end{align*}
\]

Akin to blocks in C-style languages, we sometimes use braces (instead of parenthesis) to group sequenced expressions (i.e., \(\{ e_1; \ldots; e_i; \ldots; e_n \}\)) in definitions.

To reduce an expression to a value, the expression is decomposed into an evaluation \(E\) context and a redex following the rules for forming evaluation contexts of Figure 2.1. We use \(e[x \mapsto v]\) to denote capture free substitution of \(x\) by \(v\) in \(e\). All pure expressions not involving the control operator shift are completely standard and can be reduced independently of the evaluation context. We use \(e \rightarrow e'\) to denote reduction in an evaluation context \(E\) (rule \(\text{CONG}\)) and \(e \rightarrow^* e'\) for the reflexive, transitive closure of our reduction relation.

The reduction rule for shift makes use of a special evaluation context \(H\), which we call a capture context. Like the context \(E\) the capture context is a call-by-value context, but does not include any \(\text{reset}\)-frames. To reduce a shift, we thus capture the context delimited by the closest dynamically surrounding \(\text{reset}\), reify it as a function, and bind it to \(k\). The rule \((\pm \text{shift} \pm)\) leaves us with interesting design choices: As summarized by Dybvig et al. (2007) and Shan (2004b),
2.1. Delimiting Control

Syntax:

Expressions \( e ::= e(e) \)
\( | \) \( \text{reset} \{ e \} \) application
\( | \) \( \text{shift} \{ k \Rightarrow e \} \) delimiting control effects
\( | v \) continuation capture

Values \( v ::= x \mid c \mid \lambda x. e \)

Evaluation Contexts:

\( E ::= \square \mid E(e) \mid v(E) \mid \text{reset} \{ E \} \)
\( H ::= \square \mid H(e) \mid v(H) \)

Operational Semantics:

\( (\delta) \text{ } c(v) \rightarrow \delta(c, v) \) if \( \delta(c, v) \) is defined

\( (\beta) \text{ } (\lambda x. e)(v) \rightarrow e[x \mapsto v] \)

\( \text{reset} \{ v \} \rightarrow v \)

\( e \rightarrow e' \)
\( E[e] \rightarrow E[e'] \) [CONG]

Figure 2.1. Syntax and semantics of delimited control \((\lambda dc)\). Reduction rule \((shift)\) is defined in-text.

control operators can be characterized by whether or not they leave the (outer) delimiter behind and whether or not the body of the captured continuation is enclosed by the (inner) delimiter. That is, we can choose between four different variants of the rule \((\pm shift \pm)\), each corresponding to a different control operator introduced in the literature (Dybvig et al., 2007):

\( (+shift-) \text{ } \text{reset} \{ H[shift \{ k \Rightarrow e \}] \} \rightarrow \text{reset} \{ e[k \Rightarrow \lambda x. H[x]] \} \)

Leave the outer delimiter behind, but do not include the inner delimiter in the continuation. This semantics corresponds to the control operator \(F\) by Felleisen (1988) also referred to as control in later publications.

\( (-shift-) \text{ } \text{reset} \{ H[shift \{ k \Rightarrow e \}] \} \rightarrow e[k \Rightarrow \lambda x. H[x]] \)

Neither leave the delimiter behind, nor contain it in the continuation. Shan (2004b) refers to this variant of control as control_0.

\( (-shift+) \text{ } \text{reset} \{ H[shift \{ k \Rightarrow e \}] \} \rightarrow e[k \Rightarrow \lambda x. \text{reset} \{ H[x] \}] \)

Do not leave the outer delimiter behind but include the inner delimiter in the continuation. This corresponds to the control operator shift_0 by Danvy and Filinski (1989, Appendix C).

\( (+shift+) \text{ } \text{reset} \{ H[shift \{ k \Rightarrow e \}] \} \rightarrow \text{reset} \{ e[k \Rightarrow \lambda x. \text{reset} \{ H[x] \}] \} \)

Both leave the delimiter behind and contain the delimiter in the continuation. This semantics corresponds to shift as introduced by Danvy and Filinski (1989, 1992)).
2. From Delimited Control to Effect Handlers

We use \((\pm shift\pm)\) as an abbreviation for the choice between the four different reduction rules \((+ shift-)\), \((- shift-)\), \((- shift+)\), and \((+ shift+)\). This classification of control operators has been presented by Dybvig et al. (2007), but we change their notation from \(\pm F\pm\) to \(\pm shift\pm\). Since we can instantiate it with four rules for \((\pm shift\pm)\), we also refer to \(\lambda dc\) as a calculus schema.

Delimiters and Modular Reasoning

The choice of the evaluation semantics \(\pm shift\pm\) has no influence on our particular example above. However, it makes significant difference in non-trivial programs and has implications on modular reasoning:

**The outer delimiter**  
Leaving the outer delimiter behind, rule \((+ shift\pm)\) guarantees encapsulation of control effects. That is, \(reset\{ e \}\) is guaranteed to be a pure term and thus only has “a single exit point” (Felleisen, 1988, p. 186). For a terminating expression \(e\), the code following \(reset\{ e \}\) will be executed exactly once. This property is important to reason about resource safety in presence of control effects (Felleisen, 1988). Take the following example:

```plaintext
val file = open("my-file.txt"); reset { action(); close(file)
```

No matter how \(action\) in the above example uses control effects; the \((+ shift\pm)\) variants guarantee that the file will eventually be closed, given \(action\) terminates.

**The inner delimiter**  
Similarly, delimiting the body of the continuation \((\pm shift\pm)\) guarantees that all subsequent calls to \(shift\) are conceptually delimited by the same \(reset\). This is illustrated by the following example\(^2\) adapted from Danvy and Filinski (1990):

```plaintext
def flip() = shift { k ⇒ k(true) ++ k(false) };
reset { List(flip()) || flip() }
```

Here, we use the control operator \(shift\) to express an ambiguity effect. For this example, both \((\pm shift\pm)\) variants result in the list \(List(true, true, true, false)\). Delimiting the body of the continuation is essential for type-safety of the example. Both calls to \(k\) return a list and \(flip\) returns a boolean. This is not the case for the variant \((- shift\pm)\), which would make this example ill typed. After reducing the first \(shift\) with rule \((- shift\pm)\) the example corresponds to:

```plaintext
def k(x) = {
def flip() = shift { k ⇒ k(true) ++ k(false) };
List(x || flip())
};
k(true) ++ k(false)
```

The marked call to \(flip\) will result in a stuck term, since the body of \(flip\) cannot be reduced without a corresponding delimiter. In contrast, evaluating the example with rule \((+ shift\pm)\) will

\(^2\)We use \(List(e_1, ..., e_n)\) to denote the creation of a list with \(n\) elements and the infix operator \(\pm\) for list concatenation. We use the infix operator \(x || y\) to express strict boolean disjunction.
2.1. Delimiting Control

diverge. Again, after one step of reducing shift with rule \((\_\text{shift}_-\_\) we obtain:

\[
\text{reset} \{ \\
\quad \text{def } k(x) = \\
\quad \quad \text{def } \text{flip}() = \text{shift} \{ k \Rightarrow k(\text{true}) ++ k(\text{false}) \}; \\
\quad \quad \text{List}(x || \text{flip}()) \\
\quad \}:
\]

\[
k(\text{true}) ++ k(\text{false})
\]

This time the second call to \text{flip} will be delimited. However, the captured continuation also contains the highlighted call to the first captured continuation, which in turn will call \text{flip} again, and again.

**Multiple effects with \(-\text{shift}_+\)** As it has already been observed by Danvy and Filinski (1989, Appendix C), the \(-\text{shift}_+\) variant (also called \text{shift}_0) allows to access outer contexts by shifting multiple times within the body of the shift. For example we can add a second effect operation \text{raise} to our example:

\[
\begin{align*}
\text{def } \text{raise}(msg) &= \text{shift} \{ k_{\text{flip}} \Rightarrow \text{shift} \{ k_{\text{raise}} \Rightarrow \text{None} \} \}; \\
\text{def } \text{flip}() &= \text{shift} \{ k_{\text{flip}} \Rightarrow k_{\text{flip}}(\text{true}) ++ k_{\text{flip}}(\text{false}) \}; \\
\text{def } \text{action}() &= \ldots \\
\text{reset} \{ \text{Some} (\text{reset} \{ \text{List}(\text{action}()) \}) \}
\end{align*}
\]

Here \text{action} can make use of both \text{raise} and \text{flip}. In case of \text{raise} we immediately abort with None. Compare this to

\[
\begin{align*}
\text{def } \text{raise}(msg) &= \text{shift} \{ k_{\text{raise}} \Rightarrow \text{None} \}; \\
\text{def } \text{flip}() &= \text{shift} \{ k_{\text{raised}} \Rightarrow k_{\text{flip}}(\text{true}) ++ k_{\text{flip}}(\text{false}) \} \\
\text{def } \text{action}() &= \ldots \\
\text{reset} \{ \text{List}(\text{reset} \{ \text{Some}(\text{action}()) \}) \}
\end{align*}
\]

which enumerates all possible outcomes of \text{flip} in a depth-first manner. An exception terminates the search in the current execution branch and potentially continues execution in another branch. We can define \text{action} to model a drunk trying to flip a coin (Kammar et al., 2013):

\[
\begin{align*}
\text{def } \text{action}() &= \\
\quad \text{if } (!\text{flip}()) \text{raise}("\text{dropped it}") \\
\quad \text{else if } (\text{flip}()) \text{"heas" else "tails"}
\end{align*}
\]

the first interpretation results in None, while the second one yields:

\[
\begin{align*}
\text{List}(\text{Some("heas")}, \text{Some("tails")}, \text{None})
\end{align*}
\]

By assigning a number of shifts to each effect, programs can simultaneously use different effects (exceptions and ambiguity in this example). It is this variant of delimited control, that is semantically closest to algebraic effect handlers (Kammar et al., 2013; Forster et al., 2017).

To summarize the comparison, the semantics of \(-\text{shift}–\) is very general since we can easily express all other forms by manually inserting calls to \text{reset}; the \(+\text{shift}_\pm\) variants enable effect encapsulation and reasoning about external, linear resources; the \(\pm\text{shift}_\pm\) variants guarantee that all subsequent effects are confined to the call of the continuation. The \(-\text{shift}_+\) variant allows users to combine multiple effects, albeit statically fixing the ordering of effects.
2. From Delimited Control to Effect Handlers

<table>
<thead>
<tr>
<th>One Control Operator</th>
<th>Family of Control Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg shift$</td>
<td>pushSubCont</td>
</tr>
<tr>
<td>control</td>
<td>Gunter et al. (1995)</td>
</tr>
<tr>
<td>$shift$</td>
<td>control</td>
</tr>
<tr>
<td>F</td>
<td>Dybvig et al. (2007)</td>
</tr>
<tr>
<td>$\neg shift$</td>
<td>controller</td>
</tr>
<tr>
<td>shift</td>
<td>Sitaram and Felleisen (1990)</td>
</tr>
<tr>
<td>$shift$</td>
<td>$\xi_n$</td>
</tr>
</tbody>
</table>

Figure 2.2. Comparison of different control operators in the literature.

2.2 Families of Delimited Control Operators

The semantics of $\neg shift$ allowed us to express multiple effects, such as exceptions and ambiguity, while at the same time it guaranteed that using the effects together in the same program is well-defined. However, as we have seen in the example above, the order of effects is encoded in the number of required shifts and thus statically fixed.

To overcome this issue and to express multiple effects more robustly, Sitaram and Felleisen (1990), Danvy and Filinski (1990), and Hieb and Dybvig (1990) almost simultaneously each generalized a different variant of delimited control from one control operator to a family of control operators. Figure 2.2 organizes the relevant related work, grouping by the semantics in terms of $\pm shift$ and by the generalization to families of control operators. While details of the approaches differ, conceptually the control operator $shift$ and its delimiter are generalized to a family of control operators $shift_p$ (and reset correspondingly) indexed by what we refer to as a prompt $p$. Each control operator for one particular $p$ can be used to express a different effect without risking interference between different control operators.

Figure 2.3 follows Gunter et al. (1995) to capture this indexing formally. We extend and generalize the calculus schema $\lambda_{dc}$ to $\lambda_{dcp}$ (pronounced “delimited control with prompts”). In the syntax we replace the control operator $shift$ by its indexed counterpart $shift_p$ and reset by reset$_p$, accordingly. We extend values with the set of prompts $p \in P$ and include an operation new$_p$ (Gunter et al., 1995; Dybvig et al., 2007) to create fresh prompts. Every prompt $p$ gives rise to a control operator $shift_p$ and its matching delimiter reset$_p$. Like in $\lambda_{dc}$, the reduction rule ($\pm shift$) is a rule scheme and offers four different choices for the operational semantics.

We also adjust the reduction rules accordingly. The most important change compared to the semantics in Figure 2.1 is that now the capture context $H_p$ is also indexed (Gunter et al., 1995). Just like previously omitting reset altogether in $H$, the side condition ($p \neq p'$) asserts that the innermost reset delimits the extent of the captured continuation. However, now the continuation can contain delimiters as long as they are indexed by a different prompt. The continuation captured by $shift_p$ represented by the capture context $H_p$ is thus delimited by the closest reset for the very prompt $p$.

Four variants of delimited control As in the calculus schema $\lambda_{dc}$, we again offer four variations of reduction rule ($\pm shift$). The discussion on tradeoffs of the different $\pm shift$ variants carries over from the single-prompt setting to the multi-prompt setting. However, there is one important difference. Considering the $+ shift$ variants, a delimited expression reset$_p \{ e \}$ now is only pure with respect to one particular prompt $p$. In consequence, we lose the guaranty of the single-prompt setting, where reset$_p \{ e \}$ would return exactly once. This is not the case in the
2.2. Families of Delimited Control Operators

Syntax:

Expressions \( e \ ::= \ldots \mid \text{reset}_p \{ e \} \mid \text{shift}_p \{ k \mapsto e \} \mid \text{new}_\text{prompt} \)

Values \( v \ ::= \ldots \mid p \)

Evaluation Contexts:

\[
E ::= □ \mid E(e) \mid v(E) \mid \text{reset}_p \{ E \} \\
H_p ::= □ \mid H_p(e) \mid v(H_p) \mid \text{reset}_{p'} \{ H_p \} \text{ if } p \neq p'
\]

Operational Semantics:

\[
\begin{align*}
&\quad \quad (\text{fresh}) \quad \text{new}_\text{prompt} \quad \quad \rightarrow \quad p \ \text{fresh} \\
&\quad \quad (\text{reset}) \quad \text{reset}_p \{ v \} \quad \quad \rightarrow \quad v \\
&\quad \quad (\pm \text{shift}) \quad \text{reset}_p \{ H_p[\text{shift}_p \{ k \mapsto e \} ] \} \quad \rightarrow \quad \text{reset}_p \{ e | k \mapsto \lambda x. \ \text{reset}_p \{ H_p[x] \} \} \\
\end{align*}
\]

Figure 2.3. Generalization of syntax and semantics to families of control-operators (\(\lambda_{dcp}\)).

multi-prompt setting. Other control effects (with \(p' \neq p\)) can still capture \(\text{reset}_p \{ e \}\) (and expressions following it) as part of the continuation. The following example illustrates that generalizing to multiple prompts invalidates any linearity assumptions:

\[
\text{reset}_{p'} \{ \text{val}\_file = \text{open("my - file.txt")}; \ \text{reset}_p \{ \text{action}(); \ \text{close}(\text{file}) \} \\
\]

We can now assume \text{action} to be defined as \(\text{shift}_{p'} \{ k \mapsto 0 \}\), which discards the continuation \(k\) and aborts the computation up to \(\text{reset}_{p'}\). In consequence, \(\text{reset}_p \{ \text{action}() \}\) aborts and the file will be opened, but not closed.

**Multiple prompts give rise to multiple effects** Choosing one of the variants \(\pm \text{shift}_+\), we can recast our \textit{drunkFlip} example from the last section to use prompts to distinguish different effects:

\[
\begin{align*}
\text{val exc} &= \text{new}_\text{prompt} \\
\text{val amb} &= \text{new}_\text{prompt} \\
\text{def raise}(msg) &= \text{shift}_{exc} \{ k \mapsto \text{None} \} \\
\text{def flip()} &= \text{shift}_{amb} \{ k \mapsto k(\text{true}) + + k(\text{false}) \} \\
\text{def action()} &= \ldots \\
\text{reset}_{exc} &= \text{Some}(\text{reset}_{amb} \{ \text{List}(\text{action}()) \} ) \\
\end{align*}
\]

To reorder the effects, with prompts, we can now keep the definitions of \textit{flip} and \textit{raise} unchanged. We just need to rearrange the order of the resets to:

\[
\begin{align*}
\text{reset}_{amb} \{ \text{List}(\text{reset}_{exc} \{ \text{Some}(\text{action}()) \}) \} \\
\end{align*}
\]
2. From Delimited Control to Effect Handlers

This demonstrates the flexibility gained from generalizing to multiple prompts. While before, we could express multiple effects with $\text{shift}_0$, we needed to statically assign a number of shifts to each effect and thus fix the order of effects upfront. With multiple prompts, this choice is deferred to the point of $\text{reset}$. In particular, the implementation of effect operations is independent of the ordering.

2.2.1 On the Choice of Prompts

So far, when generalizing to multiple prompts we have assumed that there exists a set of prompts $P$ whose elements can be used to index the control operators. Different approaches in the literature not only vary in the choice of $\pm \text{shift}_\pm$ but also in their treatment of prompts. Some approaches treat prompts as first-class values, while others do not. Some include an explicit $\text{new\_prompt}$ operation, whereas others make prompt creation implicit or assume a globally static set of prompts. The exact treatment of prompts is important. This can easily be seen by removing $\text{new\_prompt}$ from the language and choosing the static singleton set $P = \{ \# \}$ for the source of available prompts. In this case, the extension to multiple prompts is conservative and degenerates to $\lambda_{dc}$.

Dynamic prompts Gunter et al. (1995), Sitaram and Felleisen (1990), Hieb and Dybvig (1990), and later Dybvig et al. (2007) all allow the creation of new prompts at runtime. This way, users can dynamically create new control operators and corresponding delimiters.

Gunter et al., Sitaram and Felleisen, and Dybvig et al. additionally make prompts first class objects. Gunter et al. and Dybvig et al. introduce a separate type of prompts $P = \text{Prompt}$ and offer built-in functions $\text{new\_prompt}$ (resp. $\text{newPrompt}$) to the user to create new prompts at runtime. In contrast, Sitaram and Felleisen use natural numbers to distinguish levels of control operators, that is $P = \mathbb{N}$ and consequently do not need to offer a way to create fresh prompts.

Different to the semantics presented in Figure 2.3, their prompts form a hierarchy and the comparison on prompt equality in the capture context thus needs to be replaced by $p < p'$.

Implicit prompts In contrast to approaches with first-class prompts and explicit prompt creation, Hieb and Dybvig (1990) introduce the operator $\text{spawn} \{ c \Rightarrow e \}$ that unifies creating a fresh prompt and delimiting the extent of the continuation. Conceptually, invoking $\text{spawn}$ creates a new control operator (named $c$ for “controller”), which is delimited by this very call to $\text{spawn}$. Following Dybvig et al. (2007), the semantics of $\text{spawn}$ can be captured by

$$\text{spawn} \{ c \Rightarrow e \} \equiv \text{val } p = \text{new\_prompt}; \text{def } c(\text{body}) = \text{shift}_p \{ k \Rightarrow \text{body}(k) \}; \text{reset}_p \{ e \}$$

with semantics $\text{-shift}_+$ for the operator $\text{shift}_p$. By grouping the creation of prompts and delimiting of continuations in one feature, Hieb and Dybvig (1990) introduce a slight restriction on expressivity. Always creating a fresh prompt $p$ prevents users from using the same prompt on multiple, potentially different $\text{reset}_p$s.

Static prompts The effect operation $\text{raise}$ above illustrated that control effects can be used to model exceptions. Deliberately ignoring subtyping, to model the exceptional control flow of Java, we can instantiate $\lambda_{dcp}$ with a globally static set of prompts, corresponding to the different types of exceptions, such as:

$$P = \{ \text{ArithmeticException, NullpointerException, } \ldots \}$$
Conceptually, exception types are not first-class, though this view changes when also considering runtime reflection. Since exceptions discard the continuation, we can choose any of the $\text{-shift}_2$ variants as the reduction rule.

### 2.2.2 Static Hierarchies

Like the generalization to multiple prompts, the CPS-hierarchy (Danvy and Filinski, 1990) is an alternative approach that gives rise to a family of control operators. Danvy and Filinski (1990) translate programs with multiple levels of control operators by repeatedly applying a CPS transformation. Each iteration of the CPS transformation adds an additional continuation argument to the translated program, which gives rise to a new pair of control operators $\langle e \rangle_i$ (i.e., reset) and $\xi_i \cdot k \cdot e$ (i.e., shift). Since the number of CPS translations needs to be known at translation time, this way they obtain a static hierarchy of control operators. While in terms of delimiter treatment, their semantics corresponds to $\text{+shift}_2$, there is a significant semantic difference to our description with prompts: in the CPS-hierarchy, the control operator $\xi_n \cdot k \cdot e$ captures the first $n$ delimited contexts and binds their composition to $k$ in $e$. Likewise, $\langle e \rangle_n$ delimits the first $n$ contexts (Danvy and Filinski, 1990). As a consequence, when implementing effects like $\text{flip}$ and $\text{raise}$ as above, we need to globally fix the order of effects, for instance choosing $\xi_1$ to express $\text{flip}$ and $\xi_2$ to express raise. From a users point of view, programming in the CPS hierarchy is thus closer to our development of multiple effects with $\text{shift}_0$ ($\text{-shift}_2$) in the previous section, than to programming with multiple prompts.

### 2.2.3 Multi-Prompt Delimited Control in Effekt

In the second part of this thesis, we base our implementations of the Effekt library on multi-prompt delimited control in the semantics $\text{-shift}_2$. This holds true for both our Scala implementations in Chapters 4 and 6, as well as the Java implementation in Chapter 5. As we will see, every handler implicitly introduces a fresh prompt. Our operational semantics is thus closest to the one of $\text{spawn}$ (Hieb and Dybvig, 1990).

### 2.3 From Delimited Control to Effect Handlers

In the last two sections, we have revisited two important generalizations in the history of control operators: first from undelimited control to delimited control and then from one control operator to a family of control operators. In this section, we will now draw connections between multi-prompt delimited control and effect handlers.

It was Sitaram (1993), who performed an important (though not often recognized) step on the way from delimited control to effect handlers. Sitaram observed that with $\text{callcc}$ (and likewise with other control operators)

> “the handling of the continuation takes place at the identical site as the creation of the continuation” — Sitaram (1993, p. 148)

This is fundamentally different when programming with exception handlers. With exception handlers, $\text{throw}$ just $\text{signals}$ an exception, but the $\text{handling}$ is performed at the exception handler.

---

3We are grateful to Youyou Cong to bring the work of Sitaram (1993) to our attention.
2. From Delimited Control to Effect Handlers

Following Sitaram (1993), we can apply the same idea to control effects, and instead of
we now write:

\[
\text{reset}_p \{ \quad \text{handle}_p \{ (x, k) \Rightarrow \text{body} \} \; \text{in} \; \{
\text{shift}_p \{ k \Rightarrow \text{body} \} \;
\text{do}_p(\text{arg}) \} \}
\]

That is, instead of grouping continuation capture and handling in the construct \text{shift}, we group handling with the continuation delimiter \text{handle} (called run by Sitaram). The effect operation \text{do} (called fcontrol by Sitaram) now merely signals a control transfer to the corresponding handler. The choice of the terminology “handler” is not accidental, as the construct was inspired by exception handling (Sitaram, 1993).

Figure 2.4 defines a calculus scheme modifying \( \lambda_{dcp} \) to \( \lambda_{dch} \) (pronounced “delimited control with handlers”). As in \( \lambda_{dcp} \), prompts are first-class and can be created with \text{new}_\text{prompt}. Like in all previous presentations of calculus schemes, \( \lambda_{dch} \) offers the choice of four different reduction rules \( \pm \text{do} \). The semantics of Sitaram (1993) corresponds to \( + \text{do} \). However, as mentioned in Section 2.1, of the four different variants the reduction rule \( - \text{shift} + \) is closest to effect handlers. Similarly, we will always assume rule \( - \text{do} + \) in the remainder of this thesis unless otherwise noted.

Example  Using prompts to distinguish different effect operations, we can express our running example as follows:

\[
\begin{align*}
\text{val amb} &= \text{new}\_\text{prompt}; \\
\text{val exc} &= \text{new}\_\text{prompt}; \\
\text{def flip}() &= \text{do}_\text{amb}() \\
\text{def raise}(\text{msg}) &= \text{do}_\text{exc}(\text{msg}) \\
\text{def action}() &= \ldots
\end{align*}
\]

The effect operations \text{flip} and \text{raise} immediately forward to control operations \text{do}_\text{amb} and \text{do}_\text{exc} respectively. They do not carry any implementation details. The implementation itself is now located at the handlers:

\[
\begin{align*}
\text{def collect}(\text{action}) &= \text{handle}_\text{amb} \{ () \; k \Rightarrow k(\text{true}) \; k(\text{false}) \} \; \text{in} \; \{ \text{List}(\text{action}()) \} \\
\text{def maybe}(\text{action}) &= \text{handle}_\text{exc} \{ (\text{msg}, k) \Rightarrow \text{None} \} \; \text{in} \; \{ \text{Some}(\text{action}()) \}
\end{align*}
\]

For more flexibility, we abstract over the handlers as functions \text{collect} and \text{maybe}. Similar to our previous examples, handling the effects gives:

\[
\begin{align*}
\text{collect}(\lambda(), \text{maybe}(\text{drunkFlip})) \\
\text{maybe}(\lambda(), \text{collect}(\text{drunkFlip}))
\end{align*}
\]

We can safely treat the involved prompts and definitions of effect operations as globally static. This does not prevent us from providing alternative implementations in handlers like:

\[
\begin{align*}
\text{def report}(\text{action}) &= \text{handle}_\text{exc} \{ (\text{msg}, k) \Rightarrow \text{log}(\text{msg}) \} \; \text{in} \; \{ \text{action}(); () \}
\end{align*}
\]

\(^4\)We deliberately adjusted the syntax of Sitaram (1993) to be closer to the one of effect handlers.
2.3. From Delimited Control to Effect Handlers

Syntax:

Expressions  \( e ::= \ldots \)

| handle\(_p\) \( h \) in \{ \( e \) \} delimiting effects
| \( \text{do}_p(v) \) effect operation call

Handlers  \( h ::= \{ (x, k) ⇒ e \} \) handler clause

Evaluation Contexts:

\[
E ::= □ | E(e) | v(E) | \text{handle}_p h \text{ in } \{ E \} \\
H_p ::= □ | H_p(e) | v(H_p) | \text{handle}_{p'} h \text{ in } \{ H_p \} \quad \text{if } p \neq p'
\]

Operational Semantics:

\[
\ldots \quad \text{reset} \quad \text{handle}_p h \text{ in } \{ v \} \quad \rightarrow \quad v \\
\text{(±do±)} \quad \text{handle}_p h \text{ in } \{ H_p[\text{do}_p(v)] \} \quad \rightarrow \quad \text{handle}_p h \text{ in } \{ e[x \rightarrow v, k \rightarrow \lambda y. \text{handle}_p h \text{ in } \{ H_p[y] \} ] \} \\
\text{where } h = \{ (x, k) ⇒ e \}
\]

Figure 2.4. Syntax and semantics of multi-prompt delimited control with handlers \( (\lambda_{dch}) \).

This alternative handler for exceptions assumes logging facilities to report the provided message and always return the unit value. While programs express that they use the \textit{raise} effect, the exact semantics is determined by the handler. We can think of the handler as a \textit{dynamic binder} for the effect operation – a point of view, which we will explore in detail in Chapter 3.

Moving the handling (\textit{body}) from the control operation to the delimiter seems like a simple change in syntax. However, it has profound consequences on the way users can reason about effectful programs.

**Localized reasoning about the continuation**  With \textit{shift}_p each use of the control operator to capture the continuation can vary. Parts of the program might use \textit{shift}_p to discard the continuation and abort with a dummy value (like \textit{raise} above), other parts might use it as a backtracking facility (like \textit{flip} above) for the same delimiter. Even though different occurrences of \textit{shift}_p need to coordinate the way they use the continuation, they are spread all over the program. Not so with handlers. Locating \textit{body} with the delimiter establishes an important restriction: all bodies of \textit{shift}_p need to coincide. This way, a programmer can easily determine all usages of the continuation for a prompt \( p \) by analyzing the corresponding handler. The different ways a handler might capture and use the continuation is thus confined to one module. This allows syntactically localized reasoning about the continuation usage.

**Localized reasoning about effects**  Syntactically grouping the handlers with the delimiter has another important consequence. It naturally suggests that effects used by the expression \textit{body} are evaluated in the context of the handler, not in the context of the corresponding \textit{shift}_p or \textit{fcontrol}. What might seem obvious (when considering handlers) initially lead to confusion in
2. From Delimited Control to Effect Handlers

the context of delimited control. Danvy and Filinski (1990) describe “the desirable relation”\(^5\)

\[
\text{shift}_p \{ k \Rightarrow k(e) \} \equiv e
\]

which as they remark, in general, does not hold for the CPS-hierarchy. Translated to our setting of multiple prompts, by simple reduction we can see that it also does not hold for any of our multi-prompt \(\pm \text{shift}_{\pm}\)-variants.

\[
\text{reset}_p \{ H_p[\text{shift}_p \{ k \Rightarrow k(e) \}] \} \rightarrow^* \text{reset}_p \{ (\lambda x. \text{reset}_p \{ H_p[x] \})(e) \}
\]

In general, the reduction of expression \(e\) might involve a reduction of \(\text{shift}_{p'}\) for any other prompt \(p' \neq p\). This prompt \(p'\) in turn might be delimited in the capture context \(H_p\) since the capture context only excludes delimiters for the same prompt\(^6\). We want to argue, that this relation is not desirable after all! As illustrated above, effects in \(e\) are evaluated in the current evaluation context, while \(\text{shift}_p \{ k \Rightarrow k(e) \}\) leads to a shift in perspective: The body of \(\text{shift}_p\) is evaluated at the position of the corresponding \(\text{reset}_p\). Grouping handling expressions with the delimiter clarifies this and furthermore allows syntactically localized reasoning about the effect usage. With handlers, a programmer can lexically reason about effects used by the handler since the scope of evaluation and the scope of definition is syntactically unified. In general, this is not true for languages with \(\text{shift}_{p'}\)-like control operators.

Localized type checking Grouping delimiter and handling expressions not only helps the programmer to reason about programs, it also simplifies type checking. While type systems for various variants of control operators exist (Danvy and Filinski, 1989; Asai and Kameyama, 2007; Biernacka et al., 2011; Materzok and Biernacki, 2011) they are typically quite involved. One particular challenge is answer type modification (Asai and Kameyama, 2007). Answer type modification describes the situation where the body of a shift changes the resulting type (the answer type) at the corresponding reset. Take the following example:

\[
\text{val ans} = \text{reset} \{ 1 + \text{action()} \}
\]

Just from the type of the top-level addition, one would assume \(\text{reset}\) to return a numeric value. However, with

\[
\text{def action()} = \text{shift} \{ k \Rightarrow \lambda x. k(x) \ast 2 \}
\]

the body of \(\text{shift}\) changes the type of \(\text{ans}\) to a function between numeric values. Applying \(\text{ans}(20)\) thus yields 42. Answer type modification is powerful and can for instance express \texttt{sprintf} concisely by changing the number and the type of additional arguments based on the parsed format string (Asai and Kameyama, 2007). However, this expressiveness takes its toll on the understandability of programs and the simplicity of type checking. Handlers improve on this situation by syntactically grouping continuation usage and delimiter in one module. This simplifies type checking: Most type systems for languages with support for (algebraic) effect handlers simply require that the answer types of the handled expression and the one handling coincide (Plotkin and Pretnar, 2013; Leijen, 2017c; Lindley et al., 2017). That is, handlers do not support answer type modification. Notably, Gunter et al. (1995) achieve a similar restriction by fixing the answer type at prompt creation time.

\(^5\)For easier comparison, we present the relation in the syntax of \(\lambda_{dcp}\).

\(^6\)As before, the optional outer and inner delimiter are highlighted in grey.
2.3. From Delimited Control to Effect Handlers

Syntax:

Expressions
\[ e ::= \begin{array}{l}
    e(e) \quad \text{application} \\
    \text{handle}_{p_{h}} h \text{ in } \{ e \} \quad \text{delimiting effects} \\
    \text{do}_{p}(v) \quad \text{effect operation call} \\
    v \quad \text{value}
\end{array} \]

Values
\[ v ::= x \mid c \mid \lambda x. e \]

Handlers
\[ h ::= \{ (x, k) \Rightarrow e \} \quad \text{handler clause} \]

Evaluation Contexts:

\[ E ::= \square [ E(e) | v(E) | \text{handle}_{p_{h}} h \text{ in } \{ E \} ] \]

\[ H_{p} ::= \square [ H_{p}(e) | v(H_{p}) | \text{handle}_{p_{h}} h \text{ in } \{ H_{p} \} ] \quad \text{if } p \neq p' \]

Operational Semantics:

\[ (\delta) \quad e(v) \quad \rightarrow \delta(e, v) \quad \text{if } \delta(e, v) \text{ is defined} \]

\[ (\beta) \quad (\lambda x. e)(v) \quad \rightarrow e[x \mapsto v] \]

\[ (\text{reset}) \quad \text{handle}_{p_{h}} h \text{ in } \{ v \} \quad \rightarrow v \]

\[ (-\text{do}+) \quad \text{handle}_{p_{h}} h \text{ in } \{ H_{p} \text{[do}_{p}(v)] \} \quad \rightarrow e[x \mapsto v, k \mapsto \lambda y. \text{handle}_{p_{h}} h \text{ in } \{ H_{p}[y] \}] \quad \text{where } h = \{ (x, k) \Rightarrow e \} \]

Figure 2.5. Summary of syntax and semantics for \( \lambda_{dch} \), specialized to variant \(-\text{do}+\) and a statically declared set of prompts (adapted from Leijen, 2017c).
2.4 A Type System for Effect Handlers

In the previous section, we argued that grouping the handling expressions with the delimiter facilitates type checking. To support this observation and as a point of reference for later chapters, in this section we present a simple monomorphic type and effect system for $\lambda_{dch}$. Figure 2.6 defines the typing rules for $\lambda_{dch}$ in the variant $-do+$. Since this chapter presented the development of $\lambda_{dch}$ in multiple steps, for easier reference, we also include syntax and operational semantics (Figure 2.5). For the effect system, we build on a simplified version of the effect system of Koka (Leijen, 2017c), which we presented in similar form as $\lambda_{ach}$ in previous work (Brachthäuser and Leijen, 2019). In Koka, the set of labels is statically determined by global effect declarations, such as:

```
effect amb { flip(): bool }
```

By listing effect operations as part of an effect declaration, effect operations only implicitly refer to the corresponding effect label. Hence, labels are not first-class. Likewise, for the presentation in this section, we assume an arbitrary, but statically fixed set of prompts (that is, we exclude new_prompt from the language). Following Leijen (2017c), for every prompt $p$, we assume the presence of a unique effect declaration in $\Sigma$, that assigns a signature to the prompt ($i.e., \Sigma(p) = \tau_1 \rightarrow \tau_2$). For instance, we would require $amb \in \mathbb{P}$ and $amb : unit \rightarrow bool \in \Sigma$ to express the flip operation as:

```
def flip() { do amb() }
```

Both, the operational semantics and the effect system are adapted from Leijen (2017c). While $\lambda_{dch}$ is close to the calculus presented by Leijen (2017c), there are notable differences:

- We simplify the type system of $\lambda_{dch}$ to monomorphic types and effects.
- Handlers in $\lambda_{dch}$ do not have return-clauses.
- Handlers in $\lambda_{dch}$ are restricted to a single effect operation, whereas Leijen (2017c) allows grouping of multiple effect operations under one effect label / prompt.

Figure 2.6 introduces two typing judgments. The judgment $\Gamma \vdash_{val} v : \tau$ types values, which, by construction, cannot use any control effects. The judgment $\Gamma \vdash_{dch} e : \tau \mid \pi$ types expressions that can use control effects specified in row $\pi$. Following Leijen (2017c), a row is either empty $\langle \rangle$ or an extension with a prompt label $\langle p \mid \epsilon \rangle$. We also treat rows as equivalent up to the order of different prompts (Figure 2.6) and use the following shorthands:

```
\langle p_1, \ldots, p_n |\epsilon \rangle \equiv \langle p_1 | \ldots | p_n |\epsilon | \ldots \rangle
\langle p_1, \ldots, p_n \rangle \equiv \langle p_1 | \ldots | p_n |\{} | \ldots \rangle
```

Rule DO types control transfers do $p$ and introduces the prompt label in the effect row. In contrast, rule HANDLE eliminates the prompt label from the row to type a handler. Handlers in $\lambda_{dch}$ do not have return clauses and thus the rule HANDLE requires $e_1$ and $e_2$ both to agree on the type $\tau$. To type the handler body $e_1$, the typing environment is extended with the argument type $x : \tau_1$, and the type of the reified continuation $k : \tau_2 \rightarrow \epsilon \tau$. In particular, both the body of the handler and the continuation can use residual effects $\epsilon$, which are not handled by the handler itself.

Recently, Leijen (2018b) extended Koka with support for dynamic effect handlers by adding the rule:

```
new handle $l$ $(v)$ $\rightarrow$ handle $l$ $(v(l))$ where $l$ fresh, which is reminiscent of the above mentioned rule for spawn.
```

By generating fresh labels, the set of labels is now open and by passing the label to the handled program labels effectively become first-class objects.
2.4. A Type System for Effect Handlers

Syntax of Types:

Types \( \tau ::= \text{unit} | \text{bool} | \ldots \) set of builtin types
| \( \tau \rightarrow \epsilon \tau \) effectful functions

Effect Row \( \epsilon ::= \langle \rangle \) empty row
| \( \langle p | \epsilon \rangle \) row extension

Effect Labels \( p ::= \text{exc} \ | \text{amb} \ | \ldots \) static set of prompts

Type Environment \( \Gamma ::= \emptyset \ | \Gamma, x : \tau \)

Effect Signatures \( \Sigma ::= \emptyset \ | \Sigma, p : \tau_1 \rightarrow \tau_2 \)

Typing Rules:

\[
\begin{align*}
\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\text{val}} x : \tau} & \quad \frac{\Gamma \vdash_{\text{val}} v : \tau}{\Gamma \vdash_{\text{val}} \text{do} \ p(v) : \tau_2 | (p | \epsilon)} & \quad \frac{\Sigma(p) = \tau_1 \rightarrow \tau_2}{\Gamma \vdash_{\text{ch}} \text{e} : \tau_1 | (p | \epsilon)} \\
\frac{\Gamma, x : \tau_1 \vdash_{\text{ch}} e : \tau_2 | \epsilon}{\Gamma \vdash_{\text{val}} \lambda x. e : \tau_1 \rightarrow \tau_2} & \quad \frac{\Gamma, x : \tau_2 \vdash_{\text{ch}} \text{e}_1 : \tau_1 \rightarrow \epsilon \tau_2 | \epsilon}{\Gamma \vdash_{\text{ch}} \text{e}_2 : \tau_1 | \epsilon} & \quad \frac{\Sigma(p) = \tau_1 \rightarrow \tau_2}{\Gamma \vdash_{\text{ch}} \text{e}_1(\text{e}_2) : \tau_2} \\
\Gamma \vdash_{\text{ch}} \text{handle}_p \{ (x, k) \Rightarrow \text{e}_1 \} \text{ in } \{ \text{e}_2 \} : \tau | \epsilon}
\end{align*}
\]

Row Equivalence:

\[
\begin{align*}
\epsilon \equiv \epsilon & \quad \frac{\epsilon_1 \equiv \epsilon_2}{(p | \epsilon_1) \equiv (p | \epsilon_2)} & \quad \frac{\epsilon_1 \equiv \epsilon_2}{\epsilon_3} & \quad \frac{p_1 \neq p_2}{(p_1 | \epsilon) \equiv (p_2 | \epsilon)} & \quad \frac{p_1 \neq p_2}{(p_1 | \epsilon) \equiv (p_2 | \epsilon)}
\end{align*}
\]

Figure 2.6. Syntax of types, typing rules, and equivalence of effect rows for \( \lambda_{dch} \) (adapted from Leijen, 2017c). Every prompt \( p \) has a uniquely defined signature in \( \Sigma \).
2. From Delimited Control to Effect Handlers

2.4.1 Semantic Soundness

Since $\lambda_{dch}$ is a restriction of the language presented by Leijen (2017c), the proofs carry over mostly unchanged. Leijen (2017c) extends a previous proof (Leijen, 2014) and uses standard techniques from Wright and Felleisen (1994) to show that well-typed effectful programs cannot go wrong. That is, all effect operations are eventually handled. Here, we repeat some of the necessary lemmas and proofs of Leijen (2017c). Following Leijen, we state semantic soundness:

**Theorem 1. (Semantic Soundness)**

If $\emptyset \vdash_{dch} e : \tau | \langle \rangle$ then either $e \mapsto^* v$ where $\emptyset \vdash_{val} v : \tau$.

Note, that we require the set of effects to be empty (i.e., $\langle \rangle$). That is, the program should not have any control-effects, which are observable from the outside. We only focus on effect safety and identify the following expressions as faulty:

**Definition 1. (Unhandled effect)**

A term has an unhandled effect $p$, if it has the form $H_p[\text{do}_p(v)]$.

Leijen (2017c) continues and proves semantic soundness using two lemmas, which we prove in the remainder of this section:

**Lemma 1. (Subject reduction)**

If $\Gamma \vdash_{dch} e_1 : \tau | \epsilon$ and $e_1 \mapsto e_2$, then $\Gamma \vdash_{dch} e_2 : \tau | \epsilon$.

That is, types and effects are preserved by the reduction relation $\mapsto$.

**Lemma 2. (Effects are meaningful)**

If $\Gamma \vdash_{dch} H_p[\text{do}_p(v)] : \tau | \epsilon$, then $p \in \epsilon$.

Lemma 2 states that effect types are meaningful. Only effect handlers can remove labels $p$ from the effect row $\epsilon$.

**Proof of Semantic Soundness**

Leijen (2017c) uses Lemma 2 (stating that effects are meaningful) together with Lemma 1 (subject reduction) to prove semantic soundness (Theorem 1).

**Proof. (of Theorem 1)**

To reduce $e_1$ to a value (or diverge), we repeatedly apply $\mapsto$. Every such reduction step either leads to another expression $e_2$ or it is stuck on a faulty expression (Definition 1), that is, an unhandled effect. Lemma 1 gives us, that the new expression $e_2$ has the same type $\tau$ and the same effects $\epsilon$.

Now, let us assume that a sequence of reductions leads to an expression $H_p[\text{do}_p(v)]$, which is faulty. By subject reduction (Lemma 1), we know that $\Gamma \vdash_{dch} H_p[\text{do}_p(v)] : \tau | \epsilon$. Additionally, by Lemma 2, we know that $p \in \epsilon$.

However, the fact that $p \in \epsilon$ contradicts our assumption of Theorem 1. It requires an empty effect row. The reduction thus cannot lead to an unhandled effect.

\[ \square \]

\footnote{Leijen (2017c) allows arbitrary effects $\epsilon$, which we believe is an oversight in the original formalization.}
2.4. A Type System for Effect Handlers

Proof of Subject Reduction

To prove subject reduction (Lemma 1), we need the following two standard lemmas (Wright and Felleisen, 1994):

**Lemma 3.** (Substitution)

a. If $\Gamma, x : \tau \vdash_{\text{val}} v' : \tau'$ and $\Gamma \vdash_{\text{val}} v : \tau$ then $\Gamma \vdash_{\text{val}} v'[x \mapsto v] : \tau'$

b. If $\Gamma, x : \tau \vdash_{\text{dch}} e : \tau' | \epsilon$ and $\Gamma \vdash_{\text{val}} v : \tau$ then $\Gamma \vdash_{\text{dch}} e[x \mapsto v] : \tau' | \epsilon$

By mutual induction on the derivations $\Gamma, x : \tau \vdash_{\text{val}} v' : \tau'$, and $\Gamma, x : \tau \vdash_{\text{dch}} e : \tau' | \epsilon$.

**Lemma 4.** (Replacement)

Given a typing derivation $D$ that ends in $\Gamma \vdash_{\text{dch}} E[e] : \tau | \epsilon$, and $D'$, which is a subderivation of $D$ ending in $\Gamma' \vdash_{\text{dch}} e : \tau' | \epsilon'$ and occurs at the hole of $E$, and $\Gamma'' \vdash_{\text{dch}} e' : \tau' | \epsilon'$, then we have that $\Gamma' \vdash_{\text{dch}} E[e'] : \tau | \epsilon$.

Proof by induction on the derivation tree and case analysis of the evaluation context $E$.

Since effect operations allow to transfer control flow and send values “up the stack”, we also need the following non-standard lemma:

**Lemma 5.** (Context inversion)

If we have a typing derivation $D$ that ends in $\Gamma \vdash_{\text{dch}} E[e] : \tau | \epsilon$, and $D'$ is a subderivation of $D$ ending in $\Gamma' \vdash_{\text{dch}} e : \tau' | \epsilon'$. Then $\Gamma'$ can be weakened to $\Gamma$.

That is, the evaluation context $E$ has no influence on the type environment $\Gamma'$. Proof by induction on the derivation tree and case analysis of the evaluation context $\Gamma'$. Informally, only the rules APP and HANDLE are applicable. Neither changes the typing context.

**Proof.** (Of Lemma 1) We are now ready to prove Lemma 1. The proof proceeds by induction over reductions $\rightarrow$. We assume that the induction hypothesis holds for subterms. The interesting cases are rules $(\text{do+})$ and $(\text{reset})$:

**case $(\text{do+})$**

We need to construct a derivation for: $\Gamma \vdash e[x \mapsto v], k \mapsto \lambda y. \text{handle}_p h \in \{H_p[y]\} : \tau | \epsilon$.

From rule HANDLE we have $\Sigma(p) = \tau_1 \rightarrow \tau_2$ (1), $\Gamma \vdash_{\text{dch}} H_p[\text{do}_p(v)] : \tau | \langle p | e \rangle$ (2), and $\Gamma, x : \tau_1, k : \tau_1 \rightarrow e \vdash_{\text{dch}} e : \tau | \epsilon$ (3).

The derivation (2) includes a subderivation for $\Gamma' \vdash \text{do}_p(v) : \tau_2 | \epsilon'$ (4) with the premise $\Gamma' \vdash_{\text{val}} v : \tau_1$ (5). From (5) and Lemma 5, we can obtain $\Gamma \vdash_{\text{val}} v : \tau_1$. With (2), (4), assuming $\emptyset \vdash_{\text{dch}} y : \tau_2 | \epsilon'$, and Lemma 4, we can derive $\Gamma \vdash_{\text{dch}} H_p[y] : \tau | \langle p | e \rangle$ (6).

Using HANDLE, (1), (6), and (3) we have $\Gamma \vdash_{\text{dch}} \text{handle}_p h \in \{H_p[y]\} : \tau | \epsilon$. By LAM, we obtain $\Gamma \vdash_{\text{val}} \lambda y. \text{handle}_p h \in \{H_p[y]\} : \tau_2 \rightarrow e \tau$ (7). Assuming $x, k \not\in \text{fv}(H_p[y])$, and using Lemma 3 twice (once for $x$ and once for $k$), we finally construct the desired typing derivation.

**case $(\text{reset})$**

We need to construct a typing derivation for $\Gamma \vdash_{\text{dch}} v : \tau | \epsilon$. From rule HANDLE we have the premise $\Gamma \vdash_{\text{dch}} v : \tau | \langle p | e \rangle$ (1). Since $v$ is a value, the derivation has to end in an application of VAL with the premise $\Gamma \vdash_{\text{val}} v : \tau$ (2). From (2) we can again apply VAL choosing a different effect $\epsilon$ (instead of $\langle p | e \rangle$). This gives us the desired derivation. Intuitively, values do not have effects and we can thus choose arbitrarily.

\[\square\]
2. From Delimited Control to Effect Handlers

Proof that Effects are Meaningful

After having shown subject reduction (Lemma 1), we continue and show that faulty expressions are not typeable. To show that faulty expressions are not typeable, we need to prove Lemma 2.

Proof. (of Lemma 2) We proceed by induction over the structure of the capture context $H_p$.

**case □**

By rule $\text{Do}$, we immediately have $\epsilon = \langle p|\epsilon' \rangle$ and thus $p \in \epsilon$.

**case $H_p'(\epsilon)$**

We can apply rule $\text{App}$ to obtain the premise $\Gamma \vdash dch H_p'[\text{do}_p(v)] : \tau_1 \rightarrow \epsilon \tau_2 \mid \epsilon \,(1)$. By (1) and the induction hypothesis, we have $p \in \epsilon$.

**case $v(H_p')$**

Similar to the previous case.

**case handle$_{p',h}$ in $\{ H_p' \}$ if $p \neq p'$**

By the grammar for capture contexts, we have $p \neq p'$ (1). Applying rule $\text{HANDLE}$, we obtain $\Gamma \vdash dch H_p'[\text{do}_p(v)] : \tau | (p'|\epsilon) \,(2)$. By the induction hypothesis, (2) we have $p \in (p'|\epsilon)$. By side condition (1) we finally get $p \in \epsilon$.

\[ \square \]

2.5 Related Work and Chapter Conclusion

In this chapter, we introduced effect handlers from the perspective of control effects and discussed the relation between effect handlers and multi-prompt delimited control. In a natural progression, starting from undelimited control, via delimited control and multiple prompts, we finally arrived at effect handlers ($\lambda_{dch}$). Since this chapter summarized existing work, most related work has already been discussed throughout the text. Here, we want to briefly put $\lambda_{dch}$ into perspective of other calculi and implementations of delimited control and effect handlers.

Delimited Control and Effect Handlers

It has previously been established that algebraic effect handlers can be expressed with delimited control. Forster et al. (2017) formally relate the expressive power of effect handlers with the expressive power of single-prompt delimited control in variant $-\text{shift}+$ (that is, $\text{shift}_0$). Piróg et al. (2019) extend the work by Forster et al. (2017) to a typed setting and show that it requires type polymorphism. They provide a typed correspondence between deep handlers (Kammar et al., 2013) and $\text{shift}_0$. They also give a typed correspondence between shallow handlers, as they can be found in the language Frank (Lindley et al., 2017), and the control operator $\text{control}_0$ (that is, $-\text{shift}+$). Kiselyov and Sivaramakrishnan (2016) present a practical embedding of the language Eff into OCaml in terms of multi-prompt delimited control. They build on semantics $-\text{shift}+$ but manually delimit the body of the continuation.

Dybvig et al. (2007) give a comprehensive overview over the different control operators and show how they can be expressed in terms of $\text{withSubCont}$ (that is, semantics $-\text{shift}+$). We adopt their notation to classify control operators by delimiter treatment, but write $\pm\text{shift} \pm$ instead of $\pm F \pm$. While they present the operational semantics of control operator $\text{withSubCont}$ in terms
Comparison with Effect Handlers in Koka

As established by Forster et al. (2017), of the four different variants the reduction rule \((-\text{shift+})\) is closest to effect handlers. One can also see this relation between delimited control and algebraic effect handlers clearly when comparing the operational semantics of \(\lambda_{dch}\) with the one of Koka (Leijen, 2017c, 2018b), a language with support for algebraic effect handlers. We repeat the relevant reduction rules of Leijen (2017c) where necessary. In Koka, handlers can use the delimited continuation to provide semantics to an effect operation. The continuation capture is expressed by the following evaluation context,

\[
H_l = \square | H^l(v) | v(H^l) | \text{handle}^l_l(H^l) \quad \text{if } l \neq l'
\]

and the following reduction rule (\text{handle}):\(^{1}\)

\[
\text{handle}^l_l(H^l[\text{do}^l(v)]) \rightarrow e[x \mapsto v, \text{resume} \mapsto \lambda y. \text{handle}^l_l(H^l[y])]
\]

where \((\text{do}^l(x) \rightarrow e) \in h\)

The reduction rule shows how the call to the effect operation \(\text{do}\) is handled by the handler \(h\). Comparing this to \(\lambda_{dch}\) and mapping effect labels \(l\) to prompts \(p\), we can see that the delimited context \(H^l\) directly corresponds to the capture context \(H_p\) and that the \text{handle}-rule corresponds to rule \((-\text{do+})\), repeated here for easier comparison:

\[
\text{handle}_p h \in \{ \text{H}_p[\text{do}_p(v)] \} \rightarrow e[x \mapsto v, \text{resume} \mapsto \lambda y. \text{handle}_p h \in \{ \text{H}_p[y] \}]
\]

where \(h = \{(x, \text{resume}) \Rightarrow e\}\)

Like in rule \((-\text{do+})\), the continuation bound to \text{resume} is again delimited by \text{handle}^l_l and no outer delimiter is left behind. Also, both Koka and \(\lambda_{dch}\) perform lookup for the handler body.

However, there are also a few differences. As mentioned above, effect declarations in Koka are not limited to just one effect operation. If we also want to express multiple, different effects in \(\lambda_{dch}\) by using one prompt \(p\), we can do so by tagging the operations (Sitaram, 1993). For instance, we can express the state effect as:

\[
\text{val} \text{ state} = \text{new_prompt}
\]

\[
\text{def get()} \text{ do_state("get")}
\]

\[
\text{def set(value)} \text{ do_state("set", value)}
\]

Assigning static types to operations tagged this way would require some form of GADTs (Peyton Jones et al., 2006).

In addition to the implementation of effect operations, effect handlers in Koka also provide a "return clause". For comparison, the collect handler would be expressed as:

\[
\text{val collect} = \text{handler}\
\]

\[
\text{return} x \rightarrow \text{Cons}(x, \text{Nil})
\]

\[
\text{flip()} \rightarrow \text{append}(\text{resume(True)}, \text{resume(False)})
\]

In our examples, we inlined the return clauses. This naive solution is not always possible since the body of the return might refer to an outer handler of the same effect. Leijen (2018b) shows how to express return clauses as effect operations, which circumvents this problem.
2. From Delimited Control to Effect Handlers

2.5.1 Chapter Conclusion

In this background chapter, we presented effect handlers from the perspective of delimited control to illustrate that there are three important aspects to effect handlers:

1. One of the most important features of effect handlers is the ability to capture the continuation. We summarized existing work in $\lambda_{dc}$ and $\lambda_{dcp}$ to illustrate the design space of delimited control.

2. Effect handlers group delimiters and delimiter usage (control operator) in one module. This grouping allows localized reasoning about the usage of continuations and other control effects by effect operations. We highlight this difference by presenting both $\lambda_{dcp}$ and $\lambda_{dch}$, where the only difference is the syntactic position of the handling body.

3. With effect handlers, effect operations are dynamically bound and looked up in the evaluation context. They are thus related to dynamically scoped variables. Chapter 3 explores this connection in more detail.

With the presented calculi we not only provided necessary preliminaries for the implementations of effect handlers presented in the second part of this thesis; we also hope that our presentation puts Andrej Bauer’s statement that “effect handlers are to delimited continuations as structured programming is to goto” (at Dagstuhl Seminar 18172, March 2018) into a fresh perspective.
Chapter 3

From Dynamic Binding to Effect Handlers

In this chapter, we offer a second, alternative perspective on effect handlers: as a generalization of dynamic binding. To illustrate the relationship between effect handlers and dynamic binding, we gradually generalize dynamic binding in two steps. From ambient values, over ambient functions, to ambient control.

Ambient values are a typed implementation of dynamic binding (Moreau, 1998; Kiselyov et al., 2006). Ambient functions are bound dynamically but evaluated in the lexical scope of their binding. Ambient control further generalizes ambient functions by adding the ability to modify the control flow. It coincides exactly with effect handlers as presented in the previous chapter. Each generalization step reveals a certain aspect of effect handlers. Effect operations are dynamically bound, evaluated in the lexical scope of their binding, and can capture the continuation to resume to the call site.

We formalize all three features, ambient values, functions, and control as an extension to Moreau’s calculus of dynamic binding (1998). Unifying all language features in one framework guarantees that the interaction between ambient values, functions, and control is well defined. We also show ambient values and ambient functions can be macro expressed by ambient control. They can thus be seen as a restriction of effect handlers.

While the restricted features help us to understand effect handlers, they also serve as better programming abstraction to the user. In particular, ambient values and functions are easier to reason about than full effect handlers, while offering a similar form of effect encapsulation.

An important aspect of software development is to parametrize a software component over aspects that might vary (Parnas, 1972). Parametrizing a component with configuration, behaviour, or other components enables reuse of the same component in different contexts. Manually parametrizing components, however, can quickly become tedious, both for the component author who needs to design the component for parametrization, as well as the user of the component who needs to provide all parameters. This is especially the case for components that depend on other parametrized components. They often need to be parametrized by all transitive parameters of their dependencies. Instead of explicitly passing around parameters, we could use dynamically scoped variables.

“A dynamic variable is a variable whose association with a value exists only within the dynamic extent of an expression, that is, while control remains within that expression. If several associations exist for the same variable at the same time, the latest one takes effect. Such association is called a dynamic binding.”

— Kiselyov et al. (2006, p. 26)

3. From Dynamic Binding to Effect Handlers

In contrast to lexically scoped variables, the meaning of a dynamically bound variable is not determined in its lexical scope, but in its *dynamic scope*. In particular, lambda abstractions do not close over dynamically bound variables. In this thesis, we use *dynamic binding* (Moreau, 1998) exclusively to describe dynamically scoped variables. This is not to be confused with dynamic binding of method implementations in OOP, for which we use *late binding* (Suzuki, 1981). Due to the overloaded meaning of both *dynamic* and *variable*, in this chapter, we prefer to use the term *ambient value* to refer to a typed discipline of (immutable) dynamic variables.

Ambient values are a lightweight language feature that can be used to parametrize software components. Instead of manually parametrizing a component with additional arguments, component authors directly refer to the ambient value. Similarly, component users do not have to explicitly provide the parameters. Instead, conceptually, those parameters are bound and passed *implicitly*. This is useful in many practical situations, ranging from passing a type environment in a compiler, maintaining context information like input-positions in parsers, to associating the current request object in a web server implementation (Kiselyov et al., 2006).

Dynamic binding has a somewhat bad reputation – and not without a reason. In an untyped, unrestricted setting (as in the original Lisp (McCarthy, 1960)) one might bind dynamic variables accidentally. One might even forget to provide a binding, at all. We therefore follow the discipline of Lewis et al. (2000) who introduce *implicit parameters* as “dynamically bound values with explicit typing”. Likewise, we track ambient values in the type of a function to inform callers that they might (accidentally or not) bind them. It also helps us guarantee that all ambients are eventually bound.

The structure of this chapter follows the generalization from ambient values to ambient control, that is, effect handlers. We gradually generalize ambient values in three steps, offering new programming abstractions and insight into effect handlers:

**Ambient values**

We introduce ambient values and give examples in Section 3.1. Restricting us to ambient values highlights the aspect of effect handlers that effect operations are dynamically bound.

**Ambient functions**

The main contribution of this chapter is ambient functions: these are *bound dynamically but evaluated in the lexical scope of their binding* (Section 3.2). Ambient functions highlight the aspect of effect handlers that effect operations are evaluated at the handler (the binding site) not at the call site. We show how this small change from ambient values improves abstraction. In particular, we show how ambient functions allow users to provide precise interfaces without leaking the (side) effects of any particular implementation into the types.

**Ambient control**

We then generalize the notion of ambient functions to *ambient control*. Just like effect handlers, ambient control allows modifying the control flow. This is essential to implement control operations like exceptions and backtracking search (Section 3.3). Ambient control highlights the aspect of effect handlers that effect operations can capture the continuation to express advanced control structures – an aspect that has been the focus of the previous chapter.

We use the calculus of dynamic binding $\lambda_{db}$ as defined by Moreau (1998) in its revised form by Kiselyov, Shan, and Sabry (2006) to formalize the semantics of ambient values (which coincides exactly). We then extend it, with ambient functions as calculus $\lambda_{dbf}$, and with ambient control as $\lambda_{dbc}$. After having introduced the different language features, we present a type system, extended with *ambient rows*, that guarantees that all ambients are eventually bound.
(Section 3.4). We give a type- and semantics preserving macro translation (Felleisen, 1990) of ambient values and functions to ambient control (Section 3.5). Ambient control ($\lambda_{dbc}$) coincides exactly with effect handlers of the previous chapter ($\lambda_{dch}$). The two calculi only differ in syntax (Section 3.6). Nevertheless, we chose to present them separately to emphasize the respective aspect we started with: delimited control and dynamic binding.

Even though we can translate ambient values and functions to ambient control (and effect handlers), we argue that each individual concept merits study. Following the principle of least power (Berners-Lee, 2005), each feature gradually adds expressiveness to ambient values. Explicitly distinguishing between ambient values, functions, and control makes it (a) easier to reason about programs – for example, a program only using ambient values does not alter the control flow, (b) easier to efficiently implement – compilers can use the additional information to generate code specialized to the more limited control flow, (c) easier to learn the new concepts – users can incrementally understand the generalizations from ambient values, via ambient functions, to ambient control. We thus believe that the incremental generalization from ambient values to control offers a novel and more approachable way of introducing effect handlers.

We implemented the language design presented in this chapter in the Koka language (Leijen, 2017c), which we use to provide code examples. The implementation is available at:

https://github.com/koka-lang/koka

Koka is a strict functional language that tracks (side) effects in the types (including exceptions and divergence). The reason to use Koka is two-fold: it already has a type system based on row-types, which we adapted to track ambient bindings, and the run-time system supports effect handlers, which we use to implement ambient control. However, the ideas described in this chapter also apply to other programming languages and are not tied to Koka.

3.1 Ambient Values

To illustrate the use ambient values, we repeat an example of the canonical paper on implicit parameters by Lewis, Launchbury, Meijer, and Shields (2000). For the example, we assume that we are writing a pretty printing library that produces pretty strings from documents. Our library has the following top-level entry-point:

```haskell
fun pretty(d: doc): string
```

Unfortunately, deep inside the code, it has a hard-coded maximum display width (i.e., 40), which we would like to make configurable to use our pretty printer in different contexts with different display width.

```haskell
... if (line.length ≤ 40) then ...
```

We have two choices to abstract over the maximum display width. We can either turn the width into a global (mutable) variable, or add an extra explicit width parameter and thread it around manually. Both options come with limitations. Global mutable state is difficult to encapsulate and test, passing extra parameters requires significant change to the code base.
3. From Dynamic Binding to Effect Handlers

However, with an ambient value we can solve this cleanly. First, we declare the type signature of an ambient value at the module level:

```haskell
ambient val width: int
```

This way, we can simply refer to the ambient value `width` inside our library implementation:

```haskell
... if (line.length <= width) then ...
```

The type system of Koka tracks the use of such ambient values in row types denoted between angle brackets. In particular, the new inferred type of the pretty printing function is:

```haskell
fun pretty(d: doc): <width> string
```

The type expresses that `pretty` can only be used in a context, which provides a binding for the ambient value `width`. We can provide such a binding as follows:

```haskell
fun pretty-thin(d: doc): <> string {
  with val width = 40 in pretty(d)
}
```

Here, for the dynamic extent of evaluating the body expression `pretty(d)`, the ambient value `width` is dynamically bound to 40. Importantly, the scope of the `with`-binder is not limited lexically. Note, how the inferred type of `pretty-thin` (that is, `<> string`) now reflects that there are no more unbound ambient values.

In our implementation, we also offer a statement form of the `with`-expression that scopes over the rest of the current block. Using this form we can write the previous example as function `pretty-thin` on the left:

```haskell
fun pretty-thin(d) {
  with val width = 40;
  pretty(d)
}
```

```haskell
fun pretty-wide(d: doc): <width> string {
  with val width = width + 2;
  pretty(d)
}
```

Of course, we can also re-bind (that is, shadow) ambient values. For example, function `pretty-wide` (on the right) pretty prints part of a document doubling the current display width. The type of `pretty-wide` reflects that, even though it binds `width`, it still depends on a `width` binding in its own context.

3.1.1 Ambient Values Operationally

To formalize the calculus of ambient values, Figure 3.1 repeats the calculus of dynamic binding ($\lambda db$) by Moreau (1998) as presented by Kiselyov et al. (2006), slightly adapted to our setting. There is one difference to the presentation of Kiselyov et al. (2006): To ease comparison with calculi presented in the previous chapter, the operational semantics of ambient values uses a special evaluation context $H_p$ (Gunter et al., 1995) instead of a side condition $p \notin bp(E)$ on rule (``deal'') (Kiselyov et al., 2006). Otherwise, our presentation mostly differs in formatting. In this section, we present the operational semantics. Section 3.4 then presents the type-system.
31. Ambient Values

Syntax:

Expressions \( e \ ::= \ e(e) \) application
\[\] \( \mid \) with \( b_p \) in \( \{ e \} \) dynamic binding
\[\] \( \mid p \) ambient name
\[\] \( \mid v \) value

Binding \( b_p \ ::= \) val \( p = v \) ambient value binding

Values \( v \ ::= x \mid c \mid \lambda x. e \)

Evaluation Contexts:

\( E \ ::= \square \mid E(e) \mid \sigma(E) \mid \) with \( b_p \) in \( \{ E \} \)
\( H_p \ ::= \square \mid H_p(e) \mid \sigma(H_p) \mid \) with \( b_{p'} \) in \( \{ H_p \} \) if \( p \neq p' \)

Operational Semantics:

\( \delta \) \( c(v) \) \( \rightarrow \delta(c, v) \) if \( \delta(c, v) \) is defined
\( \beta \) \( (\lambda x. e)(v) \) \( \rightarrow e[x \mapsto v] \)
\( \text{dret} \) \( \) with \( b_p \) in \( \{ v \} \) \( \rightarrow v \)
\( \text{dval} \) \( \) with val \( p = v \) in \( \{ H_p[p] \} \) \( \rightarrow \) with val \( p = v \) in \( \{ H_p[v] \} \)

\[\text{CONG} \]

\[E[e] \mapsto E[e'] \]

\( e \rightarrow e' \)

---

**Figure 3.1.** Language of ambient values, \( \lambda_{db} \) (adapted from Kiselyov et al., 2006).

Following Kiselyov et al. (2006), we assume two disjoint sets of variables: lexical variables denoted by \( x \), and ambient names (dynamic variables) denoted by \( p \). Note that ambient names are not values, they are effectful expressions. Meta variables \( b_p \), ranging over binders, are indexed by the name of the ambient \( p \) they bind. The evaluation context \( E \) contains the clause with val \( p = v \) in \( \{ E \} \). This illustrates that dynamic bindings are part of the evaluation context (that is, the stack). The right-hand-side of an ambient value binding is syntactically restricted to be a value. This way, the bound expression is guaranteed to be fully evaluated at the definition site before it is bound.

The operational semantics is given by the four rules for the relation \( \rightarrow \). Like in the previous chapter, the relation \( \mapsto \) lets us evaluate according to the evaluation context. The most important reduction rule (dval) captures the semantics of ambient values. It dynamically resolves an ambient binding \( p \) to its bound value \( v \) by searching in the evaluation context \( E \). Similar to capture contexts in Section 2.2, the special evaluation context \( H_p \) is a restriction on contexts \( E \) and excludes other binders for \( p \) (Gunter et al., 1995). Ambient values thus resolve to the closest with-binding that dynamically surrounds the ambient value. In the reduction rule (dval), we highlight the unmodified context of the call site to the ambient value \( p \). This emphasize will become important in the following sections when discussion the generalizations of ambient values and ambient control.
3. From Dynamic Binding to Effect Handlers

3.1.2 Ambient Values Summarized

Let us summarize some key aspects of ambient values.

Declared on the module level

Ambient values have to be declared, prior to their use. The declaration `ambient val p: τ` brings both the term name `p` and an equally named type into scope. We use the latter to track ambients as part of the type of a function (Section 3.4).

Resolved by name

Ambient values are resolved by name, not by type. This is similar to implicit parameters as proposed by Lewis et al. (2000) and to their implementation in Haskell. It is different to implicits in Scala (Odersky et al., 2017), which are resolved by type (see Section 3.7).

Dynamically scoped

Ambient values are dynamically scoped. Thus, lambdas do not close over bindings. This is illustrated by the following example:

```scala
with val width = 40;
val x = 1;
val f = fun() { width + x }
```

The lambda `f` is created in a context on the left where `x` is bound to `1` and `width` is bound to `40`. Nevertheless, it only closes over the value of `x`, not over `width`. This also becomes visible in its inferred type `() → <width> int`. Hence, running the example results in `61`.

Type- and effect checked

The usage of ambients is tracked in the type of functions. The type and effect system guarantees, that all ambients are eventually bound. Section 3.4 presents a type and effect system.

Ambient types are inferred

While the ambient value `width` could be used in some deeply nested helper function of the pretty-printing library, we neither have to change the call to that helper function, nor to any other transitive caller. Also, by inferring the types, we do not need to change any of the function’s type signatures. Being able to infer types is important for maintainability.

The following example again illustrates scoping of ambient bindings.

```scala
with val width = 40;
val g = fun() { with val width = 80 in width + 1 };
val h = with val width = 80 in (fun() { width + 1 });
println(g());
println(h())
```

Here we bind two functions `g` and `h`, and invoke them. The first `println(g())` prints `81`, since during the evaluation of `g()` the ambient value `width` is bound to its innermost binding of `80`. However, `println(h())` prints `41` instead: during evaluation of `h()`, the `width` is bound dynamically to `40` – the binding to `80` was only present during the evaluation of the function value as such. This illustrates that functions do not close over ambient values.
3.2 Ambient Functions

Of course we can bind function values as ambient values. We refer to those dynamically bound function values as ambient lambdas. For example, we can declare the ambient value

\[
ambient\ val\ \text{emit-naive}: (s: \text{string}) \to <> ()
\]

and use the ambient lambda somewhere in our pretty-printing library to emit partial output:

\[
...
\text{match}(doc)\ {\text{Text}(s) \to \text{emit-naive}(s)}
\]

Unfortunately, the ambient value declaration has a type signature that severely limits the possible implementations. For example, an implementation might want to use the display width:

\[
\text{with val width = 80;}
// \text{type error: emit-naive cannot use ambient value \text{`width'}}
\text{with val emit-naive = (fun(s: \text{string})\ {println(s.truncate(width))})}}
\]

This leads to a type error, since the type signature of \text{emit-naive} promises to return unit without using any ambient values (\(<> ()\)). To work around this restriction, we can of course change the type signature of the value declaration to:

\[
ambient\ val\ \text{emit-naive} : ((s : \text{string}) \to <\text{width}> ())
\]

But now the signature exposes accidental details of one particular implementation. Even worse, it might not be the desired semantics, since the width would be dynamically bound at the call site of \text{emit-naive}, while we often want to bind it at the definition site and encapsulate this implementation detail in the definition of \text{emit-naive}.

**Ambient Values and Closure** In this particular example, we could lookup the ambient value once, bind it to an explicit value and close over that value. In fact, the desired semantics is related to closure: the binding should be resolved at the definition site of the function, not the call site. However, not only the ambient bindings, which an ambient lambda uses, are resolved at the call site – the same also extends to other (control) effects. For example, \text{emit-naive} might append the output to a file as:

\[
\text{with val emit-naive = (fun(s)\ { ... width ... append(file, s)})}}
\]

The function \text{append} uses the \text{io} effect. In Koka (Leijen, 2014), this is reflected in the type of the defined function (i.e., \((\text{string}) \to <\text{width}, \text{io}> ()\)). Hence, just as before, this definition of the ambient value \text{emit-naive} does not coincide with the declaration and is rejected by the type checker.
3. From Dynamic Binding to Effect Handlers

3.2.1 Ambient Functions: Dynamic Binding with Lexical Scoping

As motivated above, we would like to encapsulate implementation effects at the definition site of a function. Using the traditional closure semantics suffices for simple values, but does not scale to more general (control) effects. What we need instead is our new feature of ambient functions, which we summarize by:

Ambient functions are **dynamically bound**, but **statically evaluated** in the lexical context of their binding.

Similar to ambient values, we first declare an ambient function as:

```plaintext
ambient fun emit(s: string ): ()
```

Like with ambient values, the use of ambient functions is reflected in the type of an effectful program. For example, if we call `emit(...)` somewhere in our pretty-printing library, the inferred type of `pretty` becomes:

```plaintext
fun pretty(d: doc): <width, emit> string
```

The type now reflects that the function depends on ambient bindings for both the display width and an emitter function. Ambient functions are bound similar to ambient values. We can change our implementation of `pretty-thin` to also provide a binding for `emit`:

```plaintext
fun pretty-thin(d) {
    with val width = 40;
    with fun emit(s) { println(s.truncate(width)) };
    pretty(d)
}
```

This definition of `emit` is very different from the previous binding as an ambient value `emit-naive`. Like before, `emit` is dynamically bound and can be used for the extent of evaluating `pretty(d)`. However, when the ambient function `emit` is called, its body now executes in the lexical context of its definition and not in its calling context. In particular, the ambient value `width` in the body of `emit` now always resolves to `40` even if `width` is dynamically rebound inside `pretty(d)`.

Importantly, the fact that the implementation of `emit` uses the ambient value `width` is encapsulated at the call site. It neither leaks into the type of `emit` nor into its callers. Since `emit` additionally uses `println` (i.e., the console effect), the inferred return type of `pretty-thin` is `<console> string`.

3.2.2 Ambient Functions Operationally

Figure 3.2 extends the syntax and operational semantics of $\lambda_{db}$ to $\lambda_{dbf}$ (pronounced “dynamic binding with functions”) to add support for ambient functions. In contrast to the ambient value rule (deal), in rule (dfun) we first evaluate the function body $e$ outside the calling evaluation context $H_p$. That is, the function body is evaluated in the scope of its definition. Again, we highlight the original call-site context to emphasize this. After evaluating the body, we resume in the original calling context with the result bound to a fresh variable $y$. While in the calculus we distinguish between calls to ambient lambdas (i.e., $(p)(e)$) and ambient function calls (i.e., $p(e)$), we make no such difference in the surface syntax of our Koka implementation. However,
3.2. Ambient Functions

**Extended Syntax:**

Expressions

\[ e ::= \ldots \]

| \( p(v) \) | ambient call

Binding

\[ b_p ::= \ldots \]

| \( \text{fun } p(x) \{ e \} \) | ambient function binding

**Extended Evaluation Contexts:**

\[ E ::= \ldots | p(E) \]

\[ H_p ::= \ldots | p^\prime(H_p) \]

**Extended Operational Semantics:**

\[ (dfun) \ 	ext{ with fun } p(x) \{ e \} \text{ in } \{ H_p[p(v)] \} \rightarrow (\lambda y. \text{with fun } p(x) \{ e \} \text{ in } \{ H_p[y] \})(e[x \mapsto v]) \]

**Figure 3.2.** Extension of \( \lambda db \) with ambient functions (\( \lambda dbf \)).

We require the ambient signatures (declared at the top level of a module) to be globally unique. This way, the compiler (and users) can statically determine whether a call refers to an ambient function or a lambda bound by an ambient value by inspecting the corresponding signatures.

It is important to highlight the difference between the call to an ambient function and an ambient lambda. To see this difference, let us compare rule \( (dfun) \) with the following reduction using \( (dval) \) followed by \( (\beta) \):

\[
\begin{align*}
\text{with val } p = \lambda x. e \text{ in } \{ H_p[(p)(v)] \} & \rightarrow \text{with val } p = \lambda x. e \text{ in } \{ H_p[(\lambda x. e)(v)] \} \\
& \rightarrow \text{with val } p = \lambda x. e \text{ in } \{ H_p[e[x \mapsto v]] \}
\end{align*}
\]

The only difference between the above reduction of an ambient lambda and rule \( (dfun) \) is the context in which the expression \( e \) is evaluated. For ambient lambdas it is context of the call site (that is, with val \( p = \lambda x. e \text{ in } \{ H_p[\square] \} \)). In contrast, the body of ambient functions is evaluated in is context of the definition site (that is, \( (\lambda y. \ldots)(\square) \)). This small difference in operational semantics makes a big difference in terms of encapsulation.

3.2.3 A Novel Abstraction Mechanism

Changing the evaluation context to the defining scope seems like a minor extension, but it has profound implications on abstraction. In particular, we are now able to confine ambient bindings and other used effects to the lexical context of the ambient function definition. Continuing with our example, we may want to collect all generated output using local mutable state:

```plaintext
fun emit-collect(action) {
    var out := "";
    with fun emit(s) { out := out + s + "\n" } in action();
    out
}
```

47
3. From Dynamic Binding to Effect Handlers

The dynamically bound function `emit` can access the locally scoped mutable variable `out` from its body and update it. As we will see in Section 3.3.4, our local mutable variables are stack allocated and thus should not be accessed outside of their dynamic scope. In general, variable access can escape the scope through closures (Osvald et al., 2016). Similarly, if we would bind the `emit-naive` function as an ambient value as in

```plaintext
with val emit-naive = (fun(s) { out := out + s + "\n" }) in action()
```

a reference to the `out` variable might indirectly escape through `emit-naive`. To prevent this, Koka not only tracks ambients but also the use of mutable state and other effects as part of the type. Thus, the Koka type checker rejects this example due to the restrictive ambient declaration of `emit-naive`, which does not admit mutable state.

In contrast, the body of ambient functions executes in the lexical context of their definition and our example type-checks. Any implementation related effects, such as the use of `out`, are encapsulated in the scope of `emit-collect`. At the same time, in `action`, the ambient function `emit` can be used as a function of type `string → <emit> ()`. For example, the expression

```plaintext
emit-collect(fun() { emit("hello"); emit("world") })
```

just returns the string "\nworld\n" without any side effects observable from the outside.

Closures and mutable variables Readers with a background in C#, JavaScript, or Scala would expect this very behavior. Those languages inhabit a middle ground: lambda expressions can capture local variables by reference and would thus behave like in our example. While they encapsulate the state-effect, they still leak other effects, like throwing an exception, to the calling context.

Example: Depth-First Traversal

To illustrate the abstraction power of ambient functions, we adapt the example of Lewis, Launchbury, Meijer, and Shields (2000). The authors describe a depth-first traversal of a graph (King and Launchbury, 1995), where the auxiliary function `dfs-loop` is implicitly parameterized by three functions: one to mark vertices, one to query if a vertex is marked, and one to get the children of a vertex in the graph. In the original example, the authors then use `runST` (Peyton Jones and Launchbury, 1995) to efficiently implement the marking with isolated mutable state. Figure 3.3 translates the example to Koka, using ambient functions.

The `dfs` function completely encapsulates the use of (scoped) mutable state to efficiently implement the marking of the visited vertices. The inferred type of `dfs-loop` reflects that it only depends on the declared ambient functions and has no other effects:

```plaintext
fun dfs-loop(vs : list<vertex>) : <children, marked, mark> list<rose>
```

In contrast, Lewis et al. (2000) bind functions as `values` and thus leak the effects (that is, mutable state) of one particular implementation into the type of the `dfs-loop` function. The loop needs to be written in a monadic style and has the type:

```plaintext
dfsLoop :: (?children :: Graph → [Vertex],
    ?marked :: Vertex → ST s Bool,
    ?mark :: Vertex → ST s ()) ⇒ [Vertex] → ST s [Rose]
```

```plaintext
```

48


3.3. Ambient Control

```plaintext
alias vertex = int
type graph { ... }
type rose { Rose(v: vertex, sub: list<rose> ) }
ambient fun marked(v: vertex): bool
ambient fun mark(v: vertex ): ()
ambient fun children(v: vertex ): list<vertex>
fun dfs(g: graph, vs: list<vertex>): list<rose> {
  var visited := vector(g.bound, False);
  with fun children(v) { g.gchildren(v) };
  with fun marked(v) { visited[v] };
  with fun mark(v) { visited[v] := True }; dfs-loop(vs)
}
fun dfs-loop(vs: list<vertex>) { match(vs) {
  Nil → Nil
  Cons(v, rest) →
    if (marked(v)) then { dfs-loop(rest) }
    else { mark(v); Cons(Rose(v, dfs-loop(children(v))), dfs-loop(rest)) }
  }
}
```

*Figure 3.3.* Implementation of a depth-first traversal (adapted from Lewis et al., 2000). Call sites of ambient functions highlighted.

Here the ST effect, used in the implementation of the operations, leaks into the definition of `dfsLoop`. Note also how the type of `dfsLoop` is quite verbose. With implicit parameters, the names of implicits (like `?children`) are not declared on the top level. This is more flexible but also leads to larger type signatures, as it is the case for `dfsLoop`. With explicit type declarations for ambient values, the types are more concise and allow simplified type checking at the use site of an ambient function.

Ambient functions encapsulate used effects as implementation details. Encapsulation is particularly relevant when combining ambient functions with general control effects, like exceptions. To express control effects, in the next section, we thus perform a last step of generalization.

3.3 Ambient Control

On our journey to effect handlers, we started with ambient values, which are dynamically scoped. We continued with the generalization of ambient functions, which encapsulate effects at the definition site. We have seen the interaction between stack-allocated mutable state and ambient functions and alluded to the fact that ambient functions integrate well with other control effects, like exceptions. However, so far our language does not support control effects.

To also express general control effects in our unified framework, we perform one further step of generalization: from ambient functions to *ambient control*. Ambient functions are evaluated in their defining lexical scope. But like regular functions they return to the calling context. Ambient control additionally *returns* to the defining lexical scope – similar to how exceptions “return” to their innermost `try` block.
3. From Dynamic Binding to Effect Handlers

3.3.1 Aborting Control

To motivate ambient control, we follow Leijen (2016) and use ambient functions and ambient control to implement parsers. First, we declare the two ambients next and fail:

```plaintext
ambient fun next(): char
ambient control fail(msg: string): a
```

Using the two ambients, we can readily write a parser that expects a certain character in the input stream or fails otherwise:

```plaintext
fun expect(e: char): <next, fail> char {
    val c = next();
    if (c == e) then c else fail("Expected " + e + " but got " + c)
}
```

Similarly, the parser that exactly accepts the word "hello" can be expressed by:

```plaintext
fun hello(): <next, fail> () {
    expect('h'); expect('e'); expect('l'); expect('l'); expect('o'); ()
}
```

As we will see, syntactic differences notwithstanding, ambient control is identical to effect handlers from Section 2.3. It is thus not surprising, that ambient control also subsumes exception handling. In our case, fail corresponds to throw and an ambient control binding corresponds to a try handler. For example, here is a function that transforms an exception throwing action to a maybe<a> result:

```plaintext
fun to-maybe(action: () → <fail|e> a): e maybe<a> {
    with control fail(msg) { Nothing }
    Just(action())
}
```

The syntax maybe<a> denotes type constructor application to the universally quantified type parameter a.

**A note on effect polymorphism** Koka implements effect (and ambient) polymorphism in terms of row polymorphism (Leijen, 2005, 2014, 2017c). In fact, the use of ambients is just one particular class of effects that we track in the types. In the above example, action can thus use arbitrary other effects e besides the fail effect. The binder to-maybe binds and removes the ambient fail from the list of effects, while the resulting program still uses effects e. The syntax <p|e> denotes row extension with the abbreviations defined in Section 2.4. Additionally, e is short for the empty row extension <|e>. All one character type- and row-variables implicitly introduce a universal quantification. The full type signature of to-maybe thus is: forall<a, e>. (action: () → <fail|e> a) → <|e> maybe<a>.}
3.3. Ambient Control

To run the parser, we also need to define a binding for the ambient function `next`:

```haskell
fun reader(input: string, action: () → <next, fail|e> a): <fail|e> (a, string) {
  var cs := input.list;
  val res = with fun next() {
    match(cs) {
      Nil → fail("Unexpected EOS")
      Cons(c,cc) → { cs := cc; c }
    }
  }
  in action();
  (res, cs.string)
}
```

Similar to `emit-collect`, the function `reader` uses scoped mutable state to keep track of the remaining characters. It binds `next` in the dynamic extent of `action` to return the next character or otherwise uses the ambient control `fail` if no such character is available. Note, that this time, we purposefully do not encapsulate the use of `fail`. The program `action` itself can also use `fail` and potentially other effects `e`.

Finally, we can use the two binder functions `to-maybe` and `reader` to check whether a given input has the string "hello" as a prefix.

```haskell
fun parse-hello(input: string): maybe<((), string)> {
  to-maybe(fun() { reader(input, fun() { hello() }) })
}
```

If `fail(...)` is called, it will directly return to its definition point with the value `Nothing` as the result of `parse-hello`. This is the case when invoking `parse-hello("help")`. In contrast, calling `parse-hello("hello world")` will successfully parse the prefix and yield `Just(((), " world"))`.

**Remark** Higher-order functions like `to-maybe` and `reader(input)` can be thought of as first-class binders for ambients – for `fail` respectively `next` in this case. In Koka, we treat those binder functions uniformly like `val`, `fun`, or `control` binders and use the following extension of the `with` statement syntax:

```haskell
fun parse-hello(input: string): maybe<((), string)> {
  with to-maybe;
  with reader(input);
  hello()
}
```

That is, if `with` is followed by an expression, i.e. not a keyword `val`, `fun`, or `control`, we treat the expression as a higher-order binder function and pass the rest of the block as a function argument (Leijen, 2018b, Sec. 4.4). Also, note the curried application of `reader`.

### 3.3.2 Resuming Control

In the previous parsing example, we defined a new ambient control `fail` to abort the parsing in case of an error condition. Using `fail` together with `next` allowed us to describe expected sequences of characters, like "hello". However, one important feature of parsers is still missing:
3. From Dynamic Binding to Effect Handlers

how can we express alternative productions in a grammar? To this end, we declare a second ambient control function choice:

    ambient control choice(): bool

We can use choice to recognize the language AsB ::= 'a' AsB | b, multiple characters a followed by a single b, with the following parser:

    fun as-b() : <fail, next, choice> int {
        if (choice()) then { expect('a'); as-b() + 1 }
        else { expect('b'); 0 }
    }

As it turns out, ambient control functions like fail and choice not only return to their defining context, but also allow us to resume at the call site. To enable this, a control binding is passed an extra (implicit) argument resume that can be used to return to the calling context. Like in the previous chapter, resume is a first-class function and represents the continuation delimited by the corresponding control binder.

Using resume, we can bind fail and choice to collect all possible parse results in a list:

    fun collect(action: () → <fail, choice|e> a): e list<a> {
        with control choice() { append(resume(True), resume(False)) }
        with control fail() { Nil }
        Cons(action(), Nil)
    }

The binder for choice immediately returns to the call site with the value True. Resuming will eventually produce a value of type a, which is wrapped in a singleton list. Since the extent of the continuation resume is delimited by the control binder for choice, both calls to resume will return a list of results. Similarly, the extent of the continuation captured by calling fail is delimited by the fail binder. The order of binders is important here: since the binder for choice is the outermost one, capturing (and discarding) the continuation at the binder for fail only aborts the current alternative with an empty list.

Finally, we can run the parser with

    fun parse-as-b(input: string): list<int> {
        with collect;
        with reader(input);
        as-b()
    }

giving Cons(3, Nil) for parse-as-b("aaab") and Nil for parse-as-b("aaac"). Importantly, we can only parse a string like "b" since local variables in reader are stack-allocated, not heap-allocated. Conceptually, the captured continuation corresponds to a segment of the runtime stack and also includes the values of mutable, stack-local variables at the time of capturing. This enables the necessary backtracking behavior, when resuming a second time with False. If we would use global (heap-allocated) state instead, the call to resume(False) would not backtrack the position of the input stream, but would continue where the (potentially failed) alternative resume(True) last left off. The same also holds when swapping the order of with collect and with reader(input). State changes are then persisted across different alternatives.
3.3. Ambient Control

Extended Syntax:

| Binding | $b_p ::= \ldots | \text{control } p(x, k) \{ e \} \quad \text{ambient control binding} |

Extended Operational Semantics:

$$(dctl) \text{ with control } p(x, k) \{ e \} \text{ in } \{ H_p[p(v)] \} \rightarrow e[x\mapsto v, k\mapsto \lambda_y. \text{ with control } p(x, k) \{ e \} \text{ in } \{ H_p[y[\ldots]\}$$

Figure 3.4. Extension of $\lambda_{dbf}$ with ambient control ($\lambda_{dbc}$).

3.3.3 Ambient Control Operationally

Figure 3.4 defines $\lambda_{dbc}$ (pronounced “dynamic binding with control”) as an extension of $\lambda_{dbf}$ with ambient control. In contrast to the surface syntax, in the calculus we explicitly bind the continuation argument $k$. For abortive ambient control binders, which discard the continuation $k$ (like to-maybe for fail), we can simplify the reduction rule ($dctl$) to:

$$(abort) \text{ with control } p(x, k) \{ e \} \text{ in } \{ H_p[p(v)] \} \text{ where } k \notin e \rightarrow (\lambda x. e)(v)$$

This derived rule is similar to the rule ($dfun$) for ambient functions except that we do not continue evaluation in the original context.

3.3.4 Mutable Variables as Ambient Control

The previous example illustrated that the interaction between local mutable state and ambient control is subtle due to the first-class (delimited) continuation captured by resume. This is already remarked upon by Moreau (1998, p. 276), who calls for “a single framework integrating continuations, side effects, and dynamic binding.”. Kiselyov et al. (2006) studied the in the context of delimited continuations.

However, it turns out we can view local mutable state in terms of ambient control and thus not need a special semantic treatment. In particular, we can use the translation by Kammar and Pretnar (2017, Fig. 7) who show how to express mutable dynamic variables in terms of algebraic effect handlers. We reuse their translation, except that we use ambient control instead of effect handlers. First, we $\alpha$-rename such that local variables are uniquely named. For every local variable $s$, we define ambient declarations (on the left) and translate lexically bound occurrences of $s$ either to $\text{get}_s$ or $\text{put}_s$ operations (on the right):

| ambient control $\text{get}_s() : \tau$ | $s \leadsto \text{get}_s()$ |
| ambient control $\text{put}_s(x : \tau) : ()$ | $s := \text{expr} \leadsto \text{put}_s(\text{expr})$ |

Every binding of a local variable $s$ of some type $\tau$ is then translated to a function application of a $\text{local}_s$ function:

$\text{var } s : \tau := \text{init}; \leadsto \text{local}_s(\text{init, fun()} \{ \ldots \})$

...
3. From Dynamic Binding to Effect Handlers

Finally, the binder function `local_s` provides semantics to the operations `get_s` and `put_s`.

```haskell
fun local_s(init: τ, action: () → get_s, put_s | e> a): e a {
  val f = {
    with control get_s() { (fun(st) { resume(st)(st) }) };
    with control put_s(x) { (fun(st) { resume() (x) }) };
    val x = action();
    (fun(st){ x })
  }
  f(init)
}
```

The main difference to the translation in Kammar and Pretnar (2017) is that we use two separate ambient control functions, while they group the two operations under a single handler. Otherwise, both translations express mutable state by returning a function that gets the current state as input. This as also essentially the way Kiselyov et al. (2006) express dynamic binding in terms of delimited control. To implement the lookup of a binding `p`, they shift to the binder with `shift_p { f ⇒ λy. f(y)(y) }` (Kiselyov et al., 2006, Fig. 3). This corresponds to the definition of `get_s`, mapping `f` to our `resume` and `y` to our `st` parameter.

Of course, this translation is useful from a semantic perspective, but in a practical implementation, we can use more efficient mechanisms. The Koka implementation directly uses mutation of variables that are (handler) stack allocated. It also ensures that the state is properly captured as part of the stack and restored on resume.

3.4 Type-Safety of Ambient Values

While our examples in Koka already used the type- and effect-system, so far, we only presented the dynamic semantics of our calculi. In this section, we start again with `λdb` and repeat the type system of Kiselyov et al. (2006). We then extend the type system with ambient-rows to assert that all ambient values are eventually bound. We show that our type system is a conservative extension of the original type system by Kiselyov et al. (2006). Finally, we extend the row-based effect system with support for ambient functions and control.

Figure 3.5 repeats the standard typing rules for `λdb`. Like with the operational semantics of the base calculus (Figure 3.1), except for formatting, the typing rules are the same as the ones of Kiselyov et al. (2006). Like Kiselyov et al., we assume a global mapping `Σ` from dynamic variables `p` to signatures `val τ`. We denote signature extension by `Σ, p : val τ` and require that the mapping is globally unique.

The evaluation can get stuck if an ambient value is not bound, and Kiselyov et al. (2006) define such terms as `bp-stuck`:

**Definition 2. (bp-stuck)**
A term is `bp-stuck` if it has the form `E[p]` where `p ∉ bp(E).

The set of bound ambient variables in a context `E` is denoted as `bp(E)` and defined by:

- `bp(□) = ∅`
- `bp(v(E)) = bp(E)`
- `bp(E(ε)) = bp(E)`
- `bp(with b_p in E) = {p} ∪ bp(E)`

Kiselyov et al. use a side condition on rule `deal` instead of the special context `H_p`. However, by induction over the evaluation context, it can be seen that `E = H_p` if and only if `p ∉ bp(E).`
3.4. Type-Safety of Ambient Values

Syntax of Types:

Types \( \tau ::= \) unit | bool | \ldots \ 
| \ \tau \rightarrow \tau \ 

Ambient Names \( p ::= \) width | next | fail | \ldots \ 

Type Environment \( \Gamma ::= \emptyset | \Gamma, x : \tau \) 

Ambient Signatures \( \Sigma ::= \emptyset | \Sigma, p : \text{val} \tau \) 

Type Rules:

\[
\begin{align*}
\Gamma(x) &= \tau & \text{[VAR]} \\
\Gamma \vdash_{db} x : \tau & \quad \text{[VAL]} \\
\Gamma, x : \tau_1 \vdash_{db} e : \tau_2 & \quad \text{[LAM]} \\
\Gamma \vdash_{db} \lambda x.e : \tau_1 \rightarrow \tau_2 & \quad \text{[APP]} \\
\Sigma(p) &= \text{val} \tau & \quad \text{[DVAL]} \\
\Gamma \vdash_{db} p : \tau & \quad \text{[DVAL]} \\
\Gamma \vdash_{db} e_1 : \tau_1 \rightarrow \tau_2 & \quad \Gamma \vdash_{db} e_2 : \tau_2 & \quad \text{[APP]} \\
\Gamma \vdash_{db} e_1(e_2) : \tau_2 & \quad \Gamma \vdash_{db} v = v \in \{ e \} : \tau_2 & \quad \text{[WVAL]} \\
\end{align*}
\]

Figure 3.5. Standard typing rules for \( \lambda_{db} \) (adapted from Kiselyov et al., 2006, Figure 2).

That is, we can equivalently define a term to be \( bp \)-stuck, if it has the form \( H_p[p] \). The definition of \( bp \)-stuck terms thus closely resembles Definition 1 of unhandled effects (Section 2.4).

Unbound ambient values notwithstanding, the type system is sound and Kiselyov et al. (2006) prove progress and preservation:

Theorem 2. (Preservation) 
If \( \Gamma \vdash_{db} e : \tau \) and \( e \mapsto e' \), then \( \Gamma \vdash_{db} e' : \tau \).

Theorem 3. (Progress) 
If \( \emptyset \vdash_{db} e : \tau \) and \( e \) is not a value and not \( bp \)-stuck, then \( e \mapsto e' \) for some term \( e' \).

3.4.1 Effect Safety for Ambient Values

While the type system by Kiselyov et al. (2006) is pleasantly simple, it does not prevent \( bp \)-stuck terms, which is left to future work by the authors. To address this issue, Figure 3.6a defines more precise type rules for \( \lambda_{db} \) that additionally track the use of ambient values. Following the original type-system of Koka (Leijen, 2014, 2017c), we annotate every function arrow with an ambient row \( \pi \). Like in Section 2.4, we introduce two typing judgments. The judgment \( \Gamma \vdash_{val} v : \tau \) to type values, and the judgment \( \Gamma \vdash_{exp} e : \tau \) to type expressions, which can use the ambients specified in row \( \pi \).

To focus on the aspect of dynamic binding, and to allow a better comparison with the type system of Kiselyov et al., our type system is monomorphic. Leijen (2005) shows how rows can
be naturally extended with polymorphism while allowing full unification, making them well suited to combine with Hindley-Milner (Milner, 1978; Hindley, 1969) style type inference. In our implementation, we macro express ambient values, functions, and control in terms of effect handlers (see Section 3.6). This way, we are able to reuse Koka’s support for type- and effect polymorphism (Leijen, 2017c).

Tracking ambients in effect rows  Following Leijen (2005), the ambient rows can contain multiple occurrences of a name and are considered equal up to the order of the ambient names in the row (see Section 2.4, Figure 2.6). As Leijen points out, allowing duplicates in rows is important for typing ambient bindings that refer themselves to the same ambient name. For example, consider typing \( (\lambda x. \text{val } p = x \in \{ e \})(p) : \tau \) with \( \Sigma(p) = \text{val } \tau_1 \) and \( \Gamma \vdash \exp e : \tau | (p|\pi) \).

\[
\begin{align*}
\Gamma \vdash \exp \lambda x. \text{val } p = x \in \{ e \} : \tau_1 & \rightarrow (p|\pi) \tau | (p|\pi) & \text{[LAM]} \\
\Sigma(p) = \text{val } \tau_1 \end{align*}
\]

which means that the first premise is typed as:

\[
\Sigma(p) = \text{val } \tau_1 \quad \Gamma, x : \tau_1 \vdash e : (p|\pi) \quad \Gamma \vdash \exp e : \tau | (p|\pi) \]

which leads to typing \( e \) with two occurrences of \( p \) in the ambient row. Having duplicates keeps the system simple and avoids the need for row constraints (Rémy, 1994).

Conservative Extension

Our type system is a conservative extension of the original type system by Kiselyov et al. (2006). If we define a simple erasure function \( \hat{} : \tau_{imp} \rightarrow \tau_{db} \) as

\[
\hat{\cdot} = \cdot \\
(\tau_1 \rightarrow \pi \tau_2) = \hat{\tau}_1 \rightarrow \hat{\tau}_2
\]

and extend it over type environments, we can then state the following lemma:

Lemma 6. (Conservative Extension)

If \( \Gamma_{\text{exp}} e : \tau | \pi \) then \( \Gamma_{\hat{\tau}} e : \hat{\tau} \).

Immediate by erasing ambient rows from the derivation and removing identity (\text{VAL}) derivations.

3.4.2 Effect Safety for Ambient Functions and Control

Figure 3.6b extends the type system of the base language \( \lambda_{db} \) (Figure 3.6a) with rules for ambient functions and ambient control. The ambient signatures \( \Sigma \) are extended with new forms \( \text{fun } \tau_1 \rightarrow \hat{\tau}_2 \) and \( \text{control } \tau_1 \rightarrow \hat{\tau}_2 \) for ambient function declarations and ambient control declarations, correspondingly. As mentioned above, we require that bindings are globally unique. That is, for a given ambient name \( p \), from the signature environment we can uniquely determine whether it is an ambient value, function, or control.
3.4. Type-Safety of Ambient Values

Syntax of Types:

Types \( \tau \) ::= \( \text{unit} \mid \text{bool} \mid \ldots \) \( \text{builtin types} \)
| \( \tau \rightarrow \pi \tau \) \( \text{effectful functions} \)

Ambient Row \( \pi \) ::= \( \langle \rangle \) \( \text{empty row} \)
| \( \langle p | \pi \rangle \) \( \text{row extension} \)

Ambient Names \( p \) ::= \( \text{width} \mid \text{next} \mid \text{fail} \mid \ldots \)

Type Environment \( \Gamma \) ::= \( \emptyset \mid \Gamma, x : \tau \)

Ambient Signatures \( \Sigma \) ::= \( \emptyset \mid \Sigma, p : \text{val} \tau \)

Type Rules:

\[
\begin{align*}
\Gamma(x) &= \tau & [\text{VAR}] \\
\Gamma \vdash \text{val} \ v : \tau & [\text{VAL}] \\
\Sigma(p) &= \text{val} \tau & [\text{DVAL}] \\
\Gamma, x : \tau_1 \vdash e : \tau_2 | \pi & [\text{LAM}] \\
\Gamma \vdash \text{exp} \ e_1 : \tau_1 | \pi & \Gamma \vdash \text{exp} \ e_2 : \tau_2 | \pi & [\text{APP}] \\
\Sigma(p) &= \text{val} \ \tau_1 & \Gamma \vdash \text{exp} \ e : \tau_1 & \Gamma \vdash \text{exp} \ e_2 : \tau_2 | \langle p | \pi \rangle & [\text{WVAL}]
\end{align*}
\]

(a) Improved type rules for \( \lambda_{db} \). We add ambient rows to ensure all ambient values are bound.

Extended Syntax of Types:

Ambient Signature \( \Sigma \) ::= \( \ldots \mid \Sigma, p : \text{fun} \ \tau_1 \rightarrow \tau_2 \mid \Sigma, p : \text{control} \ \tau_1 \rightarrow \tau_2 \)

Extended Type Rules:

\[
\begin{align*}
\Sigma(p) &= \text{fun} \ \tau_1 \rightarrow \tau_2 & \Gamma \vdash \text{val} \ v : \tau_1 & [\text{DFUN}] \\
\Gamma \vdash \text{exp} \ p(v) : \tau_2 | \langle p | \pi \rangle & [\text{DCTL}] \\
\Sigma(p) &= \text{control} \ \tau_1 \rightarrow \tau_2 & \Gamma, x : \tau_1 \vdash \text{exp} \ e_1 : \tau_2 | \pi & \Gamma \vdash \text{exp} \ e_2 : \tau_1 | \langle p | \pi \rangle & [\text{WFUN}] \\
\Gamma \vdash \text{exp} \ \text{with fun} \ p(x) \{ e_1 \} \text{ in } \{ e_2 \} : \tau | \pi & [\text{WCTL}] \\

\end{align*}
\]

(b) Extending the typing rules of Figure 3.6a to ambient functions and control.

Figure 3.6. Effect safety for ambient values, functions, and control. Ambient rows \( \pi \) are considered equivalent up to the order of the names in the row.
3. From Dynamic Binding to Effect Handlers

Translation of Terms:

\[
\begin{align*}
\llbracket x \rrbracket &= x \\
\llbracket \lambda x. e \rrbracket &= \lambda x. \llbracket e \rrbracket \\
\llbracket e(e') \rrbracket &= \llbracket e \rrbracket (\llbracket e' \rrbracket) \\
\llbracket p \rrbracket &= p() \\
\llbracket p(v) \rrbracket &= p(\llbracket v \rrbracket) \\
\llbracket \text{with } b_p \text{ in } \{ e \} \rrbracket &= \text{with } \llbracket b_p \rrbracket \text{ in } \{ \llbracket e \rrbracket \}
\end{align*}
\]

Translation of Binders:

\[
\begin{align*}
\llbracket \text{val } p = v \rrbracket &= \text{control } p(() \text{, } k) \{ k(\llbracket v \rrbracket) \} \text{ with } k \not\in \text{fv}(v) \\
\llbracket \text{fun } p(x) \{ e \} \rrbracket &= \text{control } p(x, k) \{ k(\llbracket e \rrbracket) \} \text{ with } k \not\in \text{fv}(e) \text{ and } k \neq x \\
\llbracket \text{control } p(x, k) \{ e \} \rrbracket &= \text{control } p(x, k) \{ \llbracket e \rrbracket \}
\end{align*}
\]

Translation of Evaluation Contexts:

\[
\begin{align*}
\llbracket [] \rrbracket &= [] \\
\llbracket E(e) \rrbracket &= [ E ] (\llbracket e \rrbracket) \\
\llbracket v(E) \rrbracket &= [ v ] (\llbracket E \rrbracket) \\
\llbracket \text{with } b_p \text{ in } \{ E \} \rrbracket &= \text{with } \llbracket b_p \rrbracket \text{ in } \{ \llbracket E \rrbracket \}
\end{align*}
\]

Translation of Signatures:

\[
\begin{align*}
\llbracket \emptyset \rrbracket &= \emptyset \\
\llbracket \Sigma, p : \text{val } \tau \rrbracket &= [ \Sigma ], p : \text{control } \text{unit} \rightarrow \tau \\
\llbracket \Sigma, p : \text{fun } \tau \rightarrow \tau' \rrbracket &= [ \Sigma ], p : \text{control } \tau \rightarrow \tau' \\
\llbracket \Sigma, p : \text{control } \tau \rightarrow \tau' \rrbracket &= [ \Sigma ], p : \text{control } \tau \rightarrow \tau'
\end{align*}
\]

Figure 3.7. Translating ambient values and ambient functions to ambient control. The translation is homomorphic except for the highlighted cases.

We also extend the type rules with two new rules for checking ambient functions (WFUN) and control (WCTL) binders. The rules are similar to the WVAL rule in Figure 3.6a where the ambient name \( p \) is discharged from the ambient row. The type of the resumption function \( k \) in the type rule for control is interesting: it gets an argument of type \( \tau_2 \) (the result type of the ambient \( p \)) and returns a value of \( \tau \) (the result type of the body \( e_2 \)). Since the body of the continuation is evaluated under a with binding (see \( (dctl) \)), the ambient row is just \( \pi \) without \( p \).

To show semantic soundness of the (extended) system, we perform two steps. Firstly, we show that ambient values and ambient functions are expressible in terms of ambient control. The translation is type and semantics preserving. Some minor syntactic differences notwithstanding, the target language with ambient control \( \lambda_{dbc} \) and its type system is identical to \( \lambda_{dch} \) (Section 2.4). Semantic soundness of \( \lambda_{dbc} \) and its type system follows as a corollary.
3.5 Ambient Values and Ambient Functions as Ambient Control

In this chapter, we performed two steps of generalization. First from ambient values to ambient functions, second from ambient functions to ambient control. Ambient control truly generalizes the two other features: ambient values and ambient functions are expressible (in the sense of Felleisen (1990)) in terms of ambient control. Additionally, as we will see in the next section, our language of ambient control is identical to our typed language with effect handlers $\lambda_{dch}$ (Section 2.4). Maybe unsurprisingly, the generalization from ambient values to ambient control has lead us to effect handlers – a fact that we use as an implementation strategy for ambients in Koka. The strong theoretical foundation and expressiveness of algebraic effect handlers make this an excellent target – and not without precedent, as Kammar and Pretnar (2017) already show how to translate mutable dynamic binding to algebraic effects.

3.5.1 Translation to Ambient Control

Figure 3.7 defines a translation function $\lbrack \cdot \rbrack$ from $\lambda_{dch}$ (a language with ambient values, functions, and control) to a subset that only uses ambient control. We translate values, expressions, evaluation contexts, and signatures. We do not translate types and ambient rows, since they do not distinguish between values, functions, and control.

**Translating ambient values** As can be seen from the translation of signatures, we translate ambient values to ambient control operations taking an argument of type unit. Binders for ambient values immediately resume with the bound value:

$$\lbrack \text{with val } p = v \text{ in } \{ e \} \rbrack = \text{ with control } p(()), k \{ k([v]) \} \text{ in } \{ [e] \}$$

Accordingly, the lookup of an ambient value $p$ is translated to a ambient call $p()$. A very similar translation has previously been given by Kiselyov et al. (2006, Fig. 3), who show how to express dynamic binding in terms of (multi-prompt) delimited control, and by Kammar and Pretnar (2017, Fig. 7) who show that effect handlers can express dynamically bound state.

**Translating ambient functions** Similar to ambient values, ambient function binders translate to control binders that resume after evaluating the function body:

$$\lbrack \text{with fun } p(x) \{ e_1 \} \text{ in } \{ e_2 \} \rbrack = \text{ with control } p(x, k) \{ k([e_1]) \} \text{ in } \{ [e_2] \}$$

Ambient function calls translate to ambient control calls. All other language constructs, including ambient control are translated unchanged.

3.5.2 Type Preservation

The translation preserves types:

**Theorem.** *(Type Preservation)*

If $\Gamma \vdash_{exp} e : \tau \mid \pi$, then $\lbrack \Gamma \rbrack \vdash_{exp} \lbrack e \rbrack : [\tau] \mid [\pi]$.
3. From Dynamic Binding to Effect Handlers

**Proof.** Induction over the type rules of $\lambda_{dbc}$. We only list the two relevant cases of translating ambient values and functions.

**case WFUN**
We type $\Gamma \vdash \text{fun} \ p(x) \ {e_1} \ in \ {e_2} : \tau \mid \pi$. By induction, we can assume
- $[\Sigma] \ (p) = \text{control} \ \tau_1 \rightarrow \tau_2 \ (1)$,
- $\Gamma, x : \tau_1 \vdash \text{exp} \ [e_1] : \tau_2 \mid \pi \ (2)$, and
- $\Gamma \vdash \text{exp} \ [e_2] : \tau_1 \mid \langle p | \pi \rangle \ (3)$.

We need to verify now that we can check $\Gamma \vdash \text{exp} \ [\text{fun} \ p(x) \ {e_1} \ in \ {e_2}] : \tau \mid \pi$ which equals $\Gamma \vdash \text{exp} \ [\text{fun} \ p(x, k) \ {k(\{ e_1 \})} \ in \ {\{ e_2 \}}] : \tau \mid \pi$, with $k \notin \text{fv}(e_1)$ and $k \neq x$. To use rule WCTL, we can immediately satisfy two of its premises with (1) and (3). That leaves us to derive $\Gamma, x : \tau_1, k : \tau_2 \rightarrow \pi \ \tau \vdash \text{exp} \ [k(\{ e_1 \})] : \tau \mid \pi$. We can use rule APP to check $\Gamma, \ x : \tau_1, k : \tau_2 \rightarrow \pi \ \tau \vdash \text{exp} \ [k(\{ e_1 \})] : \tau_2 \mid \pi$. Since $k \notin \text{fv}(e_1)$, we can apply weakening and it suffices to show $\Gamma, x : \tau_1 \vdash \text{exp} \ [e_1] : \tau_2 \mid \pi$, which holds by (2).

**case WVAL**
Similar to the previous case. We type $\Gamma \vdash \text{val} \ p = v \ in \ {e} : \tau_2 \mid \pi$.
Again, by induction, we can assume $[\Sigma] \ (p) = \text{control} \ \text{unit} \rightarrow \tau_1 \ (1)$, $\Gamma \vdash \text{val} \ [v] : \tau_1 \ (2)$, and $\Gamma \vdash \text{exp} \ [e] : \tau_2 \mid \langle p | \pi \rangle \ (3)$.

We need to derive $\Gamma \vdash \text{exp} \ [\text{val} \ p()] \ (k) \ {k(\{ v \})} \ in \ {\{ e \}} : \tau_2 \mid \pi$, with $k \notin \text{fv}(v)$

We can again satisfy two premises of WCTL with (1) and (3). This leaves us to show $\Gamma, k : \tau_1 \rightarrow \pi \ \tau_2 \rightarrow \text{exp} \ [k(\{ v \})] : \tau_2 \mid \pi$. Using rule APP, leaves us to check the type of the argument to the continuation $\Gamma, k : \tau_1 \rightarrow \pi \ \tau_2 \vdash \text{exp} \ [v] : \tau_1 \mid \pi$. Again, since $k \notin \text{fv}(v)$, we can check $\Gamma \vdash \text{exp} \ [v] : \tau_1 \mid \pi$, which holds by rule VAL and (2).

The cases for DVAL and DFUN follow immediately from the induction hypothesis.

3.5.3 Preservation of Semantics
This brings us to our theorem that the translation preserves semantics:

**Theorem 5.** (Semantic Soundness)
If $e \mapsto e'$ then $[\{ e \}] \mapsto^* [\{ e' \}]$

To prove this, we need the following lemmas about the translation:

**Lemma 7.** (Translation preserves free variables)
$\text{fv}(e) = \text{fv}(\{ e \})$

Proof by induction over the structure of expressions. Translating ambient values and function binders introduces a variable $k$. The side conditions $k \notin \text{fv}(v)$ and $k \notin \text{fv}(e)$ prevent capture.

**Lemma 8.** (Translation preserves contexts)
$[\ E[e] \ ] = [\ E][\ [ e ]\ ]$

Proof by induction over evaluation contexts.

**Lemma 9.** (Translation preserves bound ambients)
$[\ H_p[e] \ ] = [\ H_p][\ [ e ]\ ]$

60
3.5. Ambient Values and Ambient Functions as Ambient Control

Proof by induction over the grammar of capture contexts $H_p$. Informally, only the kind of binder changes, not the name of the ambient.

**Lemma 10.** (*Translation is substitution safe*)

$$[[ e[x \mapsto v] ]] = [[ e ]][[ x \mapsto v ]]$$

Proof by induction over the size of the expression.

**Proof.** (*Of Theorem 5*) The proof proceeds by induction over the reduction rules of $\lambda_{dbc}$.

**case (dfun)** - with fun $p(x) \{ e \} \in \{ H_p[p(v)] \} \rightarrow (\lambda y. \text{with fun } p(x) \{ e \} \in \{ H_p[y] \})(e[x \mapsto v])$

Applying the translation, we derive:

- $[[ \text{with fun } p(x) \{ e \} \in \{ H_p[p(v)] \} ]]$
  - $= \{ \text{definition of } \cdot \text{ with } k \not\in \text{fv}(e) \}$  \(\text{(1)}\)
- with control $p(x, k) \{ k(\{ e \}) \} \in \{ \{ H_p[p(v)] \} \}$
  - $= \{ \text{Lemma 8, Lemma 9} \}$
- with control $p(x, k) \{ k(\{ e \}) \} \in \{ \{ H_p \}[p(\{ v \})] \}$
  - $\rightarrow \{ \text{definition of } \cdot \} \$ (dctl) \}
- $\rightarrow \{ \text{definition of } \cdot \} \$ (dctl) \}
- $(k(\{ e \})[x \mapsto \{ v \}], k \rightarrow \lambda y. \text{with control } p(x, k) \{ k(\{ e \}) \} \in \{ \{ H_p \}[y] \})$
  - $= \{ \text{substitute} \}$
- $(\lambda y. \text{with control } p(x, k) \{ k(\{ e \}) \} \in \{ \{ H_p \}[y] \}(e)[x \mapsto \{ v \}], k \rightarrow \ldots)$
  - $= \{ (1) \text{ and Lemma 7} \}$
- $(\lambda y. \text{with control } p(x, k) \{ k(\{ e \}) \} \in \{ \{ H_p \}[y] \}(e)[x \mapsto \{ v \}])$
  - $= \{ \text{Lemma 10} \}$
- $(\lambda y. \text{with control } p(x, k) \{ k(\{ e \}) \} \in \{ \{ H_p \}[y] \}(e[x \mapsto v]))$
  - $= \{ (1) \text{ and definition of } \cdot \} \$ (dctl) \}
- $[[ \lambda y. \text{with fun } p(x) \{ e \} \in \{ H_p[y] \}](e[x \mapsto v])$

**case (dval)** - with val $x = v \in \{ H_p[p] \} \rightarrow \text{with val } x = v \in \{ H_p[v] \}$

Structurally similar to the previous case, but requires an additional application of rule ($\beta$).

- $[[ \text{with val } x = v \in \{ H_p[p] \} ]]$
  - $\ldots$
- $(\lambda y. \text{with control } p(x, k) \{ k(\{ v \}) \} \in \{ \{ H_p \}[y] \}(\{ v \}))$
  - $= \{ \text{definition of } (\beta) \text{ and substitution} \}$
- with control $p(x, k) \{ k(\{ v \}) \} \in \{ \{ H_p \}[\{ v \}] \}$
  - $= \{ \text{definition of } \cdot \} \$ (dctl) \}
- $[[ \text{with val } p = v \in \{ H_p[v] \} ]$

\(\square\)

The proof for case (dval) explains why Theorem 5 maps one step in the source program to potentially multiple reduction steps in the translated program. In particular, we could phrase the theorem more precisely: When the source program takes one reduction step, the translated program takes either one or two reduction steps.
Figure 3.8. Translating the language of ambient control ($\lambda_{\text{dbc}}$) to delimited control with handlers ($\lambda_{\text{dch}}$) and back. Both translations are fully homomorphic.
3.6 Translating to Effect Handlers and Back

We have seen that ambient values and ambient functions can be macro expressed in terms of ambient control. The translation is type- and semantics preserving. Comparing the language with only ambient control (\(\lambda_{dbc}\)) with the language of effect handlers in Section 2.4, we find that they only differ in surface syntax. To recall, for the type system of \(\lambda_{dbc}\) we assumed semantics \(-do+,\) a globally static set of prompts, and corresponding signatures. Both, operational semantics and the type system are equivalent up to syntax.

**Theorem 6.** (Macro Equivalence)
The effect-safe calculus \(\lambda_{dbc}\) (with ambient control) and effect-safe \(\lambda_{dch}\) (in variant \(-do+)\) are macro equivalent.

To capture the relation between the two calculi formally, Figure 3.8 defines translations from ambient control to effect handlers (Figure 3.8a) and back (Figure 3.8b). Both translations are fully homomorphic and only map effect handlers to ambient control, and back. The translations are inverse to each other, as can be easily seen from the translation rules. Every ambient binder for \(p\) corresponds to a handler for a given prompt. Since prompts (\(p\)) coincide with ambient names, and effect rows \(\epsilon\) with ambient rows \(\pi\) – we do not need to translate types besides renaming meta-variables for effect rows.

The translation is type and semantics preserving as can be verified by comparing the respective rules. For easier comparison, here we repeat the relevant reduction rules:

\[
\begin{align*}
\text{(dctl)} & \quad \text{with } b_p \text{ in } \{H_p[v]\} \quad \rightarrow \quad e[x \mapsto v, k \mapsto \lambda \gamma. \text{with control } p(x, k) \{e\} \text{ in } \{H_p[y]\} \\
\text{where } b_p & = \text{control } p(x, k) \{e\}
\end{align*}
\]

\[
\begin{align*}
\text{(-do+)} & \quad \text{handle}_p \ h \text{ in } \{H_p[do_p(v)]\} \quad \rightarrow \quad e[x \mapsto v, k \mapsto \lambda \gamma. \text{handle}_p \ h \text{ in } \{H_p[y]\} \\
\text{where } h & = \{ (x, k) \Rightarrow e \}
\end{align*}
\]

We also repeat the typing judgements for ambient control and effect handlers:

\[
\Sigma(p) = \text{control } \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_{\text{exp}} \ e_2 : \tau | (p|\pi) \quad \Gamma, \ x : \tau_1, \ k : \tau_2 \rightarrow \pi \mid \tau \vdash_{\text{exp}} \ e_1 : \tau | \pi
\]

\[
\frac{\Sigma(p) = \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_{\text{dch}} \ e_2 : \tau | (p|\epsilon) \quad \Gamma, \ x : \tau_1, \ k : \tau_2 \rightarrow \epsilon \mid \tau \vdash_{\text{dch}} \ e_1 : \tau | \epsilon}{\Gamma \vdash_{\text{dch}} \text{handle}_p \ \{ (x, k) \Rightarrow e_1 \} \text{ in } \{ e_2 \} : \tau | \epsilon}
\]

We have established the equivalence between the calculi, semantics soundness of \(\lambda_{dbc}\) follows as corollary.

**Corollary.** (Semantic Soundness)
If \(\emptyset \vdash_{\text{exp}} \ e : \tau | ()\) then either \(e\) diverges, or evaluates to a value \(e \mapsto^{*} v\) where \(\emptyset \vdash_{\text{val}} \ v : \tau\).

This immediately follows from the fact that typed \(\lambda_{dbc}\) is equivalent to typed \(\lambda_{dch}\) (Theorem 6) together with semantic soundness of \(\lambda_{dch}\) (Theorem 1).
3. From Dynamic Binding to Effect Handlers

3.7 Related Work and Chapter Conclusion

We briefly review work related to dynamic binding and effect handlers.

**Implicit parameters** Ambient values are perhaps most closely related to *implicit parameters* as described by Lewis, Launchbury, Meijer, and Shields (2000). In particular, implicit parameters are immutable, named, and statically typed. In contrast to our approach, implicit parameters do not need to be declared and can be used and bound at any type. This is flexible, but can lead to large types (as shown in Section 3.2.3). It also delays possible type errors to the binding site. Lewis et al. (2000) show how implicit parameters can be elegantly implemented using regular parameter passing. This is similar to the dictionary passing translation of type classes (Wadler and Blott, 1989; Jones, 1992). Such translation could possibly be applied to ambient values as well, turning every member \( p \) in an ambient row into an additional parameter. In the second part of this thesis, we use the similar technique of explicit capability passing to implement effect handlers in object-oriented programming languages (Chapters 4 to 6).

**Dynamic binding and delimited control** In an untyped setting, dynamic binding first appeared in McCarthy Lisp (as a bug) (McCarthy, 1960). Modern dialects have lexical scoping but still provide dynamic binding: in Common Lisp one can use the `special` declaration (Steele Jr., 1990), and MIT Scheme has `fluid-let` bindings (Hanson, 1991). The semantics of dynamic binding was formalized by Moreau (1998). Later, Kiselyov et al. (2006) extend upon that work by giving a translation into delimited control, providing a unified framework for control effects and dynamic binding. Later, Kammar and Pretnar (2017) perform a similar translation and show how (mutable) dynamic variables can be expressed in terms of algebraic effect handlers – our translation of local mutable variables in Section 3.3.4 is based on this. Forster, Kammar, Lindley, and Pretnar (2017) show that in a untyped setting, algebraic effect handlers, delimited control, and monads, can all express each other through a local macro-translation (Felleisen, 1990) and thus all can express dynamic binding. Piróg et al. (2019) extend the results to the typed setting, which requires some form of type-polymorphism.

**Resolution by type** Instead of binding ambient values by name, an alternative is to resolve dynamic bindings implicitly based on their *type*. Implicit parameters in Scala (Oliveira and Gibbons, 2010; Odersky, 2019a; Odersky et al., 2017) are probably the most common example. Implicit parameters are declared on a method signature but provided automatically at the call site based on their type. Siek and Lumsdaine (2005) introduce system \( F^G \), which uses type based resolution for concept-based generic programming. Haskell type classes (Wadler and Blott, 1989; Jones, 1992; Kiselyov and Shan, 2004) are another instance where dictionaries are passed implicitly, based on their type. Oliveira et al. (2012) formalize a notion of implicit parameters, which are resolved based on their type, in the *implicit calculus*. The implicit calculus is interesting since implicit values are not only resolved by their type, but also referred to by their type. For example,

\[
\text{implicit 1 in implicit True in (even(?int) \\&\& ?bool)}
\]

evaluates to \( False \) where \(?\tau\) is used to implicitly look up a value of type \( \tau \). Resolving by type leads to issues with coherence, and stability under type substitution, as well as problems when combined with polymorphism (Schrijvers et al., 2019). Ambient values in contrast, are resolved
by name, globally declared, and have an untyped operational semantics. This way, ambient values are naturally stable under type substitution.

3.7.1 Discussion: Ambient Values and Ambient Functions

In this chapter, we introduced effect handlers by step-wise generalizing dynamic binding (that is, ambient values). We discovered ambient functions as a new language feature that resides between ambient values and ambient control. Ambient functions are a small extension to ambient values, but create new opportunities for abstraction, while avoiding the need for full generality of continuation capture. Both features, ambient values and ambient functions, are macro expressible in terms of ambient control. So, why should we consider them separately? As mentioned in the beginning of this chapter, distinguishing between ambient values, function and control has several advantages.

Learnability of concepts Users can incrementally understand the generalizations from ambient values, via ambient functions, to ambient control. In this chapter we followed these developments to open up a new perspective on effect handlers as generalization of dynamic binding. Studying the difference between lambdas bound to ambient values and ambient functions shows how effect handlers can serve as abstraction barrier, even if they do not use delimited control themselves.

Reasoning about programs A program only using ambient values and ambient functions does not alter the control flow. From the type of our example function pretty (Section 3.1), we can immediately see that it only uses ambient values. The following code example uses this:

```haskell
val f = open("output.txt");
val res = pretty(doc);
try-write(f, res);
close(f)
```

The effect row of pretty only mentions the ambient width. Since pretty only uses ambient values, it cannot alter the control flow and thus will exactly return once with a string value. Assuming that the function try-write also does not have control effects, cannot raise exceptions, and does not diverge\(^9\), we can be sure that the file handle \(f\) will be closed. This would still be true if pretty would use ambient functions, as long as the binders of those ambient functions do not use control-flow altering effects themselves.

Efficient implementation Compilers can use the additional information to generate code specialized to the more limited control flow. For ambient values and ambient functions we know that the continuation will be used in a tail-position (also called tail resumptive by Leijen (2017b)). In this case, we do not need to capture the continuation at all, which can lead to performance improvements (Leijen, 2017b). Koka previously already performed similar optimizations (Leijen, 2018a, Sec. 5) but needed to infer this information from the handler definitions. With ambient values and functions this information is readily present in the ambient declarations. The optimization thus applies naturally to our implementation of ambient values and ambient functions in Koka, as well.

\(^9\)Divergence is conservatively approximated by the Koka effect system (Leijen, 2014). If it cannot determine that a recursive function is structurally recursive, it adds the \texttt{div} effect to the function’s effect row.
3. From Dynamic Binding to Effect Handlers

3.7.2 Conclusion

In this chapter, we step-wise generalized ambient values. Each generalization illustrated an important aspect of effect handlers. Starting from ambient values, we revisited existing work on dynamic binding. The operational semantics of $\lambda_{db}$ (Figure 3.1) and its simple type system (Figure 3.5), are based on the DB calculus by Kiselyov et al. (2006). Ambient values emphasize the fact that the implementation of an effect operation is dynamically bound.

In a second step, we generalized ambient values to the novel feature of ambient functions, which in turn distils yet another aspect of effect handlers: the implementation of an effect operation is dynamically bound, but \textit{statically evaluated} in the context of its binding. The expressiveness of ambient functions resides between ambient values and ambient control (that is, effect handlers). Unlike ambient control, ambient functions do not obtain access to the continuation. In consequence, they cannot alter the control flow. However, like effect handlers, ambient functions encapsulate implementation effects at the definition site. Ambient functions thus support abstraction similar to effect handlers, without the conceptually challenging aspect of delimited control.

The third and last generalization, ambient control, is equivalent to effect handlers as presented in the previous chapter (Section 3.6). We gave a translation of ambient values and ambient functions to ambient control (Section 3.5). By macro expressing ambient values and ambient functions in terms of ambient control, we were able to reuse the soundness proof of Section 2.4.1, which in turn is an adaption of the proof for Koka by Leijen (2017c). The fact, that delimited control can express dynamic binding is not new (Kiselyov et al., 2006), the same holds for effect handlers (Kammar et al., 2013). To implement ambients in the Koka language, we use a similar translation from ambient values to effect handlers.

In the second part of this thesis, we explore an alternative to dynamic binding of effect operations: we employ \textit{explicit capability passing} to embed effect handlers in object-oriented programming languages. That is, instead of dynamically searching for the respective handler in the evaluation context, user programs are required to explicitly pass handler instances.
Part II.

Effect Handlers and Object-Oriented Programming
Effect handlers are a program-structuring paradigm with rising popularity in the functional programming language community and can express many advanced control flow abstractions. In this chapter, we present Effekt: the first design of effect handlers that revolves around object-oriented programming. Our design maps effect handler abstractions to key abstraction of object-oriented programming languages.

Different to most other existing languages with effect handlers, Effekt is centered on (explicit) capability-passing style as an alternative to dynamic binding. Capability-passing style integrates well with the paradigm of object-oriented programming.

We explore the combination of effect handlers and object-oriented programming and describe new potential to structure effectful programs and effect handlers, enabled by reusing existing abstraction. We capture the different dimensions of extensibility in a variant of Wadler’s expression problem (Wadler, 1998), which we call the effect expression problem.

Effect handlers, as introduced in the first part of this thesis, have been conceived in the setting of functional programming languages. Given this heritage, it is not surprising that effect handlers have been implemented as libraries for functional programming languages like Haskell (Kammar et al., 2013; Kiselyov et al., 2013; Wu and Schrijvers, 2015), Idris (Brady, 2013), or OCaml (Kammar et al., 2013; Kiselyov and Sivaramakrishnan, 2016). Similarly, most standalone languages that support effect handlers are following the functional programming paradigm. Examples include Eff (Bauer and Pretnar, 2015), Koka (Leijen, 2014), Frank (Lindley et al., 2017), Links (Hillerström et al., 2017), and Helium (Biernacki et al., 2019).

Implementations in the realm of object-oriented programming (OOP) are much more difficult to find. No implementation, that we are aware of, explicitly uses OOP features in its API (see discussion of related work in Section 4.7.3). Instead, library implementations like Eff in Scala (Torreborre, 2016) often re-implement the work by Kiselyov and Ishii (2015) and neglect the object-oriented nature of their host language. In this chapter, we present Effekt – the first design for effect handlers that revolves around object-oriented programming. In particular, we directly
4. Effekt – A Library Design

map the abstractions of effect handlers to key abstractions in object-oriented programming, establishing the mantra of our design:

Programming with effect handlers in Effekt is object-oriented programming.

In particular, effect signatures are interfaces declaring the effect operations, effect handlers are classes implementing those interfaces and effectful programs are methods written against the abstraction of the interface (Figure 4.1). While in this chapter we present Effekt as a Scala library, the high-level concepts are independent of Scala and applicable to every object-oriented programming language with interfaces, classes, and simple generics. In the next chapter, we present an implementation of Effekt in the language Java.

Directly mapping effect handlers to OOP features has several advantages:

Economy of concepts
Reusing existing concepts to structure effect handlers and effectful programs lets users focus on what is new with effect handlers. At the same time users can immediately apply their knowledge and intuition about interfaces and classes to structure effect signatures and handlers, correspondingly.

Reuse of abstractions
As we will see, reusing existing abstractions from OOP allows users to combine the advantages of those abstractions with the abstraction of effect handlers. The combination of effect handlers with object-oriented features enables novel modularization strategies, both for effectful programs and for effect handler implementations. For instance, with our design, users can use standard OOP techniques like inheritance, subtyping, private state, dynamic dispatch, and others to structure and implement effect handlers. By embedding handlers in a language like Scala, users can additionally use more advanced features (like mixin composition or abstract type members) to modularly describe effects and handlers.

Ease of implementation
Language implementers benefit from reusing existing abstractions since this removes the burden of re-implementing very similar abstractions all over again. Having fewer features also potentially reduces the risk of bad feature interaction. For library implementers, reuse is even more essential since it simply might not be possible to add new abstractions to a language without modifying its syntax and semantics. Our design allowed us to implement effect handlers as libraries for Scala (this chapter and Chapter 6) and Java (Chapter 5) without modifications to the languages.

To get a first impression of Effekt, consider the following piece of code written in Scala and using our library. As before in Section 2.1.1, we use the effect operation flip to nondeterministically determine whether the gambler is too drunk to catch the coin (Kammar et al., 2013). In this case, we use the effect operation raise to raise an exception. Otherwise, we return the result of a second coin toss as a string.

```scala
def drunkFlip(amb: Amb, exc: Exc): Control[String] = for {
  caught ← amb.flip()
  heads ← if (caught) amb.flip() else exc.raise("We dropped the coin.")
} yield if (heads) "Heads" else "Tails"
```

The two effect operations flip and raise are methods declared in corresponding interfaces
modeling effect signatures \texttt{Amb} (for ambiguity) and \texttt{Exc} (for exceptions). We will see the declarations of the effect signatures in the next section.

In the first part of this thesis, we revisited two important aspects of effect handlers: dynamic binding and delimited control. Both aspects can already be identified in the type signature of the method \texttt{drunkFlip}. In particular, our library design incorporates these two aspects in the following ways:

**Dynamic binding**

In contrast to most other implementations (and our presentation in Chapters 2 and 3), we do not perform a search for the implementation of effect operations at runtime. Instead, effect operations are methods on effect instances \texttt{amb} and \texttt{exc}, which \texttt{drunkFlip} receives as arguments. We refer to this alternative to dynamic binding as \textit{capability-passing style}. The terminology of \textit{capability} (Dennis and Van Horn, 1966) suggests that they entitle the holder to use the effect.

**Delimited control**

The result type of the program suggests that it does not simply compute a string, but uses control effects to do so (hence the name of the type constructor \texttt{Control}). We base our Scala library on a monadic implementation of \textit{multi-prompt delimited control} (Dybvig et al., 2007) with semantics $-\text{shift}+$ (Section 2.1.1). This is also reflected in the use of Scala’s \texttt{for}-comprehensions (Odersky et al., 2006) to sequence effectful expressions. While having to write programs in monadic style might not seem very object-oriented, it is not essential to our library design and does not prevent us from using OOP features for other aspects of the library. Chapter 5 explores an alternative implementation strategy of delimited control that lifts this limitation and enables writing programs in direct style.

Our library design revolves around the concept of \textit{capability passing}. As we will see in the remainder of this chapter, capabilities in \texttt{Effekt} encapsulate three different aspects:

- **Capabilities contain effect implementations** They give semantics to effect operations (Section 4.1). In the example program \texttt{drunkFlip}, we call effect operations as methods on the capabilities \texttt{amb} and \texttt{exc}.

- **Capabilities contain prompt markers** Effect operations can capture the continuation, which is delimited by the corresponding handler (Chapter 2). Programs written with our library have type \texttt{Control} (Section 4.2), a monadic implementation of delimited control with first-class prompts (Dybvig et al., 2007). Our capabilities contain such a prompt marker as a value member.
4. Effekt – A Library Design

Capabilities grant privileges Capabilities can be seen as constructive legitimization that the holder is entitled to use a particular effect (Dennis and Van Horn, 1966; Miller, 2006). While passing capabilities encourages developers to mention effects in the method signature, this is not enough to statically guarantee effect safety. In particular, capabilities can leak, potentially resulting in runtime errors (Osvald et al., 2016). Chapter 6 shows how we can use Scala’s advanced type system features of path-dependent types and intersection types to encode regions (Moggi and Sabry, 2001) and obtain effect safety.

In summary, the contributions of this chapter are:

- We present a library design to embed effect handlers in object-oriented programming languages. While we use Scala for our presentation, the idea of mapping effect handlers to OOP concepts is independent of Scala as a host language (Section 4.1).
- We introduce capability-passing style as an alternative to dynamically binding effect handlers. While the translation of dynamic binding to dictionary passing is not new (Lewis et al., 2000; Kammar et al., 2013), we embrace explicit passing of capabilities as part of the user facing syntax. As we will see in Chapter 6, capability passing is essential to our approach of establishing effect safety.
- We study the modularization potential newly gained by combining effect handlers and object-oriented programming. We show how reusing existing abstractions opens up new and interesting ways to modularize effect handlers. Section 4.4 relates effect handlers to the expression problem, explores several dimensions of extensibility, and shows how Effekt supports them.
- We present a novel design for stateful handlers to guarantee well-behaved interaction with continuation capture (Section 4.3). Our design is an OOP alternative to parameterized handlers that can be found in functional programming languages like Koka (Leijen, 2017c).
- We hide most of the syntactic overhead of capability passing (Section 4.6.1) by making it implicit using the Scala features of implicit parameters (Odersky, 2019a) and implicit function types (Odersky et al., 2017).

While the general design of Effekt is independent of a concrete object-oriented language, for the presentation in this thesis, we choose the programming language Dotty (in version 0.19) – a version of the Scala language that will eventually become Scala 3 (Odersky, 2018). We explain necessary Scala specifics on first encounter.

4.1 Programming with Effect Handlers in Effekt

To get a first impression of how to program with effect handlers in Effekt, and to illustrate how effect handler concepts map to object-oriented programming, let’s again look at our running example of drunkFlip.

4.1.1 Effect Signatures are Interfaces

The implementation of drunkFlip uses capabilities amb and exc, which are instances of effect signatures Amb and Exc, correspondingly. We sometimes interchangeably use the terminology
### 4.1. Programming with Effect Handlers in Effekt

```scala
trait Control[A] {
  def map[B](f: A => B): Control[B]
  def flatMap[B](f: A => Control[B]): Control[B]
  def andThen[B](c: Control[B]): Control[B]
  def pure[A](a: => A): Control[A]
  def run[A](ca: Control[A]): A
}
```

**Figure 4.2.** The control interface – a monad for multi-prompt delimited control.

**capability** and **handler instance.** While “capability” (Dennis and Van Horn, 1966) puts a focus on the concept of entitling the holder to use an effect, “handler instance” highlights the fact that handlers are implementations of effect signatures. Following our mapping of concepts from the introduction, we define effect signatures as interfaces:

```scala
trait Exc {
  def raise[A](msg: String): Control[A]
}
trait Amb {
  def flip(): Control[Boolean]
}
```

For the purpose of this section, it is enough to understand Scala traits as the equivalent to interfaces in languages like Java. The methods `raise` and `flip` model effect operations. Their return type `Control[A]` (and `Control[Boolean]`, correspondingly) suggests that they are **effectful**, that is, they can have control effects.

The concrete implementation of `Control` is not essential to the design of Effekt. For now, it suffices to understand it as some monadic type constructor with the operations summarized in Figure 4.2. In the following Section 4.2, we will go into more detail on our implementation of multi-prompt delimited control and relate it to our presentation of Chapter 2.

The methods to sequence programs without control effects (i.e. `map`) and with control effects (i.e. `flatMap`) enable us to write effectful programs in an imperative style via Scala’s `for`-comprehensions\(^\text{10}\). The function `pure` corresponds to monadic `return` (Wadler, 1995) and lifts a Scala expression without control effects into a computation. While the expression is free of control effects, it can have other side effects such as I/O. Since Scala is call-by-value, we mark the parameter (i.e., `\(\Rightarrow\) A`) as being by-name (Odersky et al., 2006) to defer those side effects to the point of running the computation by means of `run`. Using `run`, we can execute a program with control effects that computes a value of type `A` to eventually obtain that value. While we can already understand `map` as the constructive proof that `Control` is covariant in the type parameter `A` – we additionally prefix the type parameter with `+` to inform the type checker about the covariance and establish subtyping of `Control[A] <: Control[B]`, whenever `A <: B` (Odersky et al., 2006).

#### 4.1.2 Effect Handlers are Implementations

The method `drunkFlip` does not rely on any concrete implementation of the effect operations `raise` and `flip`. It merely uses the interfaces `Amb` and `Exc` in its signature. The caller is free to pick any implementations of `Amb` and `Exc`, determining the semantics of the effect operations. For

\(^{10}\)Similar to Haskell’s do-notation, `for`-comprehensions in Scala syntactically simplify writing monadic code. In general, `for { x₁ <- e₁; x₂ <- e₂; ... xn <- en } yield e` desugars to `e₁.flatMap { x₁ ⇒ e₂.flatMap { x₂ ⇒ ... en.map { xn ⇒ e } }}` (Odersky et al., 2006).
example, we can ignore control effects altogether and use native side effects to implement \texttt{Amb} and \texttt{Exc}. For \texttt{flip} we use a random number generator and for \texttt{raise} we use native exceptions\footnote{In Scala, methods with one argument can be called with braces \texttt{(pure \{ expr \})} instead of parenthesis \texttt{(pure(expr))}. This is purely syntactical and does not lead to thunking of the expression (Odersky et al., 2006).}.  

```scala
class NativeExceptions extends Exc {
  def raise[A](msg: String): Control[A] = pure { throw new RuntimeException(msg) }
}
class RandomFlip extends Amb {
  def flip(): Control[Boolean] = pure { Math.random() < 0.5 }
}
```

To run our example program, we simply pass instances of \texttt{NativeExceptions} and \texttt{RandomFlip} to \texttt{drunkFlip}, which will randomly result in either a runtime exception or one of the two possible string values.

```scala
run { drunkFlip(new RandomFlip(), new NativeExceptions()) }
```

Following our mapping of concepts, the classes \texttt{NativeExceptions} and \texttt{RandomFlip} are effect handlers: they provide semantics to the effect operations by implementing the effect signatures. However, to do so, they immediately return a pure expression without any control effects. Again, for our purposes, pure expressions do not have control-effects – they can have other side effects.

In the remainder of this section, we will pick up our handler implementations of Chapter 2 that did use control effects. We will explore an interpretation of programs, which use \texttt{Exc} and compute some result of type \texttt{R}, into programs that return an \texttt{Option[R]}. Calling \texttt{raise} will immediately abort the program with \texttt{None}. We will also explore an interpretation of programs, which use \texttt{flip} to compute a result of type \texttt{R}, into programs that return a \texttt{List[R]} by enumerating and collecting all possible outcomes.

### 4.1.3 Handling with Control Effects

We now show how to freely mix handlers if we implement them using our \texttt{Effekt} library. This is an example of the modularity benefits provided by effect handlers. The implementation of our handlers will be given shortly. For now, let us assume the following definitions:

```scala
class Maybe[R] extends Exc with Handler[R, Option[R]] { ... }
class Collect[R] extends Amb with Handler[R, List[R]] { ... }
```

Like our first interpretation of \texttt{Amb} and \texttt{Exc} above, the classes \texttt{Maybe} and \texttt{Collect} again implement the corresponding effect signatures. This time, however, the effect handlers also extend our library trait \texttt{Handler[R, E]} (defined in Figure 4.3b). In general, every handler for an effect gives semantics to the corresponding effect operations. To do so, it interprets an effectful program, which would compute a result of type \texttt{R} (mnemonic for \texttt{return type}) into a new semantic domain of type \texttt{E}, the effect domain. When implementing effect handlers, programmers choose the effect domain to have enough structure to implement the effect operations.
4.1. Programming with Effect Handlers in Effekt

Handlers introduce capabilities Before looking at the details of how the handlers are implemented, it is instructive to understand how they can be used. Recall our notion of effect handlers in Section 2.3:

```scala
handle[Amb]{ ...} in {
  handle[Exc]{ ...} in {
    drunkFlip()
  }
}
```

In comparison, using Effekt, handling effects Amb and Exc looks like\(^{12}\):

```scala
val res1: Control[List[Option[String]]] =
  new Collect handle { amb ⇒
    new Maybe handle { exc ⇒
      drunkFlip(amb, exc)
    }
  }
```

The lambda, which is passed as body to the handler `new Collect().handle(amb ⇒ ...)`, represents the dynamic scope in which the capability amb can be used. Besides delimiting the dynamic scope, `handle` also passes the handler instance unmodified as argument to the body. Thus, in our example the variable `amb` will be bound to the instance of `Collect` created on the very same line. This is also reflected by our usage of the singleton type `this.type` (Odersky and Zenger, 2005b) as type of the argument (see Figure 4.3b). We only use the singleton type for better type inference, such that `amb` can be used as a capability of type Amb without requiring manual type annotations.

Having handled both effects, running `drunkFlip` now yields for `run { res1 }`:

▶ List(Some("Heads"), Some("Tails"), None)

As in Section 2.2, we can easily swap the two handlers to obtain a different semantics where an exception leads to a termination of the search.

```scala
val res2: Control[Option[List[String]]] =
  new Maybe handle { exc ⇒
    new Collect handle { amb ⇒
      drunkFlip(amb, exc)
    }
  }
```

Since at least one of the control paths raises an exception, running `drunkFlip` with the effects handled in the different order yields for `run { res2 }`:

▶ None

\(^{12}\)Scala methods with only one argument can be used as an infix operators. Also type parameters can be omitted, if they can be inferred. Hence, handling with the Maybe handler could also be written more explicitly as `new Maybe[String]().handle(exc ⇒ ...)`. 
4. Effekt – A Library Design

```scala
class Maybe[R] extends Exc with Handler[R, Option[R]] {
  def unit(result: R) = Some(result)
  def raise(msg: String) = use { resume ⇒ pure(None) }
}
class Collect[R] extends Amb with Handler[R, List[R]] {
  def unit(result: R) = List(result)
  def flip() = use { resume ⇒ for {
    ts ← resume(true)
    fs ← resume(false)
  } yield ts ++ fs }
}
```

(a) The two effect handlers `Maybe` and `Collect` utilizing `use` to capture the continuation.

```scala
trait Handler[R, E] {
  // Subclasses need to specify how to convert a result into the effect domain
  protected def unit(result: R): Control[E]
  // Handler implementations can use this method to capture the continuation
  protected def use[A](body: (A ⇒ Control[E]) ⇒ Control[E]): Control[A] = ...
  // Handler users can use this method to delimit the scope of the continuation
  def handle(prog: this.type ⇒ Control[R]): Control[E] = ...
}
```

(b) The handler interface – the essence of the effect handler library. Handlers interpret programs of type `R` into an effect domain of type `E`. Implementations of `use` and `handle` given in Figure 4.8.

Figure 4.3. Implementation of effect handlers for `Exc` and `Amb` using the library class `Handler`.

This shows the power of effect handlers: effectful programs can be written fully agnostic of both the semantic domain into which the effects will be interpreted, as well as the order in which the effects will be handled. Importantly, changing the order of handlers changes the result type. A program that uses `Exc` returns an optional and a program that uses `Amb` returns a list. We did not have to change the effectful program that uses the effects. Notably, both, its type and its implementation can remain unchanged.

We could also implement the example without support for effect handlers. However, to write programs that use both effects (like `drunkFlip`), we would need to decide and fix upfront whether raising an exception terminates the search for possible outcomes and thus returns `Option[List[R]]`, or whether it only terminates one branch in our search and thus returns `List[Option[R]]`. Revising decisions like this in retrospect can be costly, since they potentially affect the whole codebase. Additionally, if we want to use programs, which use only one of `Exc` and `Amb`, together with programs that use both, we have to manually lift (Liang et al., 1995) the programs that use only one of the two effects. Effect handlers address these problems.
4.1. Programming with Effect Handlers in Effekt

4.1.4 Implementing Effect Handlers

Having seen how effects can be handled, we will now take a closer look at how handlers are implemented. In Figure 4.3a, we use the Effekt library to give the implementation of the handlers Maybe and Collect. The handlers extend the corresponding effect signatures (Exc and Amb), as well as the library trait Handler (Figure 4.3b).

Every handler needs to implement the abstract method unit to specify how handled programs, which do not use the corresponding effect, are lifted from R into the effect domain E. The method unit is also known as the ”return-clause” in the literature on effect handlers (Bauer and Pretnar, 2015; Kammar et al., 2013). Additionally, a handler needs to implement all the effect operations, which are specified in the effect signature. To implement the effect operations, handlers are able to utilize the instance method use, as provided by the library trait Handler. Specialized to the handler Maybe, the method use has type:

```scala
def use[A](body: (A ⇒ Control[Option[R]]) ⇒ Control[Option[R]]): Control[A]
```

In our implementation of raise, calling use { resume ⇒ ... } captures the continuation and binds it to the identifier resume in the provided body. To implement the expected semantics of exceptions unwinding the stack, we discard the continuation and immediately return None.

The handler for ambiguity, in turn, captures the continuation and invokes it twice. Once with true and once with false, each time yielding a list of possible results. Finally, it concatenates the two lists. It is important to stress again that this is only possible, because the continuation captured by use is delimited by the corresponding call to handle and the handler Collect defines the effect domain to have the type List[R].

The methods unit and use of the handler interface are marked as protected to signal that they are implementation details of the handler and should not be called from the outside.

4.1.5 Design of the Effekt Library

We designed Effekt as a library for object-oriented languages around object-oriented idioms. In summary, the Effekt library is based on the following design decisions.

Effect signatures are interfaces and handlers are classes implementing the interfaces. We establish a capability-passing style where users explicitly pass handler instances to programs that use effect operations. As we will discuss in Section 4.4, this corresponds to a shallow embedding of effect operations and is similar to the use of type classes by Kammar et al. (2013).

Handler implementations capture the continuation by explicitly calling use. In Effekt, tail resumptive handler implementations, which exclusively invoke the continuation in tail position, can be expressed as simple methods that do not capture the continuation at all.

Effekt is a library and neither changes the language nor the type system. Handlers and the operations use and handle are designed to not require an advanced type system. In particular, different to approaches based on monad transformers (Liang et al., 1995; Jones, 1995), Effekt does not require type constructor polymorphism. The usage of the singleton type this.type in the signature of handle is not essential and can be removed at the cost of type inference or some more boilerplate.

The design of Effekt, as presented in this chapter, does not establish an effect typing discipline. Users need to make sure that an effect handler is only used in the dynamic extent of a corresponding call to handle. Invoking use outside of the corresponding handler scope will result in a runtime error. We address this issue in Chapter 6.
4. Effekt – A Library Design

```scala
trait Prompt[Result] {}

def reset[R](prog: Prompt[R] ⇒ Control[R]): Control[R]
def shift[A, R](prompt: Prompt[R])(body: (A ⇒ Control[R]) ⇒ Control[A]): Control[A]
```

Figure 4.4. The prompt interface – control operators `shift` / `reset` and the marker trait `Prompt`.

4.2 Delimited Control

Our Scala implementation of Effekt implements effect handlers in terms of a monad for multi-prompt delimited control. In this section, we present a simplified version of our implementation of this monad as a specialization of the one presented by Dybvig et al. (2007). The monad `Control[A]` is very close to the monad for multi-prompt delimited control by Dybvig et al. (2007), but we specialize the exposed interface to better fit effect handlers. While Dybvig et al. present a very general framework with semantics `−shift−` that allows for the implementation of different control operators, we build on `−shift+` as our control operator of choice. This means that the body of the captured continuation is always delimited by a `reset`. As highlighted by Kammar et al. (2013), `−shift+` matches the semantics of deep handlers, where the same effect is already handled in the continuation. It additionally avoids the problem of accumulating delimiters, as observed by Dybvig et al. (2007). Section 4.7 offers a closer comparison of our implementation with the one by Dybvig et al. (2007). The version of our monad for delimited control that we introduce in this section is answer-type-safe, but it is not effect-safe. In Chapter 6, we show how to establish effect safety.

We briefly repeat the developments of Chapter 2, this time in Scala and using our monad `Control`. Consequently, the structure of the following sections is very similar: we first recapitulate delimited control, generalize to families of control operators with multiple prompts, and finally introduce the abstraction of effect handlers.

4.2.1 Delimiting Control

To recall, the following example program, which we adapt from Danvy and Filinski (1990), uses the control operator `shift` and delimiter `reset` of Section 2.1.1 in variant `−shift+`:

```
1 + reset { 10 + shift { k ⇒ k(k(100)) } }
```

The control operator `shift` captures the continuation and binds it to `k`. The continuation is delimited: only the evaluation context up to the enclosing `reset` is captured and thus the continuation corresponds to `10 + □`. This example reduces in the following steps:

```
1 + reset { 10 + shift { k ⇒ k(k(100)) } }
1 + reset { k(k(100)) }
1 + k(reset { 10 + 100 })
1 + k(110)
1 + reset { 10 + 110 }
1 + 120
121
```

The continuation `k` does not contain the frame `1 + □`, which is outside of the delimiting `reset`. From the reduction steps, we can recognize the semantics `−shift+`: The body of the continuation `k` is again delimited by `reset`, while the body of `shift` (i.e. `k(k(100)))` is not.
Example 1

Using the `Control` monad, we can translate the above example to Scala:

```scala
val ex: Control[Int] = reset { p ⇒
    shift(p) { k ⇒ k(100) flatMap { x ⇒ k(x) } map { y ⇒ 10 + y } }
} map { z ⇒ 1 + z }
```

Figure 4.2 already introduced the first part of the interface of the monad for delimited control. To actually capture continuations and delimit their scope, Figure 4.4 now additionally introduces the type `Prompt[Result]` and two functions `shift` and `reset`. The delimiter function `reset { p ⇒ PROG }` introduces a fresh prompt `p` and delimits the control effects for `p` in the provided program `PROG`. Our control operator `shift` takes a prompt as a parameter. This allows us to select which `reset` we want to shift to. It captures the current continuation up to the corresponding `reset` and passes the continuation to the body. The current example only uses one prompt `p` but we will shortly see how to utilize this additional expressivity.

### 4.2.2 Families of Delimited Control Operators

Every call to `reset` introduces a fresh prompt `p`, or in other words, each prompt `p` labels the corresponding `reset`. This gives rise to a dynamic number (that is, a family) of control operators `shift(p)`, one for each prompt created by a `reset`. The following example illustrates the use of multiple resets:

Example 2

We use `reset` twice, introducing two different prompts `p_1` and `p_2`.

```scala
val ex_2: Control[Int] = reset { p_1 ⇒
    reset { p_2 ⇒
        shift(p_1) { k ⇒ pure(21) }
    } map { if (_) 1 else 2 }
} map { 2 * _ }
```

The captured continuation `k` contains the program segment delimited by prompt `p_1`. It corresponds to the evaluation context if `(reset_{p_2} { □ }) 1` else `2`. In this example the body of `shift` discards the continuation `k` and immediately returns `21`. Hence, `run { ex_2 }` evaluates to `42`.

### 4.2.3 Answer Type Safety

Operationally, `shift(p)(k ⇒ PROG)` replaces the corresponding `reset` by the body `PROG`. To be type safe, the return type of the body thus has to match the answer type at the `reset`. In our setting of multiple first-class prompts, we guarantee this by following Dybvig et al. (2007) (respectively Gunter et al. (1995)) and parameterize prompts over the answer type `R`. In Example 2, the two prompts have type `p_1: Prompt[Int]` and `p_2: Prompt[Boolean].

Furthermore, the type of `reset[R]` requires three types to be `R`: the answer type of the created prompt (i.e., `Prompt[R]`), the result of the given program (i.e., `Control[R]`), and the return type of `reset`. Similarly, the type of `shift` uses the answer type of the given prompt and requires that (a) the return type of the continuation and (b) the return type of the given body agree.
4. Effekt – A Library Design

![Figure 4.5](four.osf)

(a) Programming with delimited control.
(b) Structured programming with delimited control.
(c) Programming with effect handlers.

**Figure 4.5.** Programming with effect handlers as structured programming with delimited control.

with (c) the answer type expected at the `reset` that introduced the prompt. Answer type safety is especially important in the presence of multiple prompts. Each reset potentially introduces a prompt with a different answer type. Shifting to a reset with the wrong type should be statically rejected. For instance, shifting to \( p_2 \) would render the example type incorrect, since this would require the body of shift to return a computation of type `Boolean`, not `Int`.

### 4.2.4 Structured Programming with Delimited Control

Multi-prompt delimited continuations are implementation internals that Effekt does not expose to the user. However, to highlight the point of Chapter 2, that programming with effect handlers is structured programming with delimited control (Kammar et al., 2013), we express our running example in terms of the operators `shift` and `reset`. Figure 4.5 provides an overview of the developments, where Figure 4.5c corresponds to programming with effect handlers and Figure 4.5b depicts the style of structured programming, which we will employ in the rest of this section. It is instructive to compare the different ways of implementing effects, since structured programming with delimited control immediately gives rise to capability-passing style.

**Example 3**

Figure 4.6 translates the handlers of our running example to directly use control effects. Let us assume the type aliases for effectful programs with the signatures of `flip` and `raise` (Figure 4.6a). The user program from the introduction then carries over almost unchanged.

```python
def drunkFlip(raise: Exc, flip: Amb): Control[String] = for {
    caught ← flip()
    heads ← if (caught) flip() else raise("We dropped the coin.")
} yield if (heads) "Heads" else "Tails"
```

The program `drunkFlip` now takes effectful functions `raise` and `flip` directly as parameters. The effectful functions directly correspond to effect operations. While the program was previously written in capability-passing style, it is now written in a very similar *operation-passing style*.

We implement what could be viewed as handlers for `raise` (Figure 4.6b) and `flip` (Figure 4.6c) as higher order functions that construct implementations of effect operations and pass them to the given program. We refer to those functions as *handler functions*. They also change the result type \( R \) to the effect domain `Option[R]` and `List[R]` respectively, just as we have previously seen. The pattern to directly implement effect handlers in terms of delimited control shows that handler functions encapsulate three aspects of effect handling in one module:
4.2. Delimited Control

- **Type declarations:**
  - `type Amb = {} ⇒ Control[Boolean]
  - `type Exc = String ⇒ Control[Nothing]

(a) Effect signatures for exception and ambiguity as type aliases for effectful functions.

```scala
def maybe[R](prog: Exc ⇒ Control[R]) = reset { p ⇒
  val raise: Exc = msg ⇒ shift(p) { resume ⇒ pure(None) }
  prog(raise) map { x ⇒ Some(x) }
}
```

(b) Handler function for the exception effect.

```scala
def collect[R](prog: Amb ⇒ Control[R]) = reset { p ⇒
  val flip: Amb = {} ⇒ shift(p) { resume ⇒ for {
    xs ← resume(true)
    ys ← resume(false)
  } yield xs ++ ys }
  prog(flip) map { x ⇒ List(x) }
}
```

(c) Handler function for the ambiguity effect.

Figure 4.6. Using answer-type-safe delimited control to declare and handle exception and ambiguity effects.

1. The handler function uses `reset` to delimit the scope of the captured continuation.
2. It *locally* uses the fresh prompt introduced by `reset` to implement the effect operations in terms of `shift`. The effect operations close over the prompt and are thus the only way to capture the continuation.
3. It finally lifts the return type of the handled function `R` into the effect domain, which makes it the answer type of the `reset`.

Grouping these aspects of effect handling in one module, it is possible to locally reason about type safety. The implementation of `raise` is only safe because we statically know from the type of `p: Prompt[Option[R]]` that the answer type expected at the `reset` is `Option[R]`. Likewise, in `collect` we use the fact that we statically know that the answer type in the body of `shift(p)` is `List[R]` to safely concatenate the results of the two calls of the continuation `resume`.

As in the introduction, we can use both handler functions in a different order to run the program, getting different results of different type.

```scala
val res1: List[Option[String]] = run {
  collect { flip ⇒ maybe { raise ⇒ drunkFlip(r, f) } }
}
val res2: Option[List[String]] = run {
  maybe { raise ⇒ collect { flip ⇒ drunkFlip(r, f) } }
}
```

The operations `raise` and `flip` are first-class functions and close over the fresh prompts `p` that we introduced with `reset`. Writing effect handlers in this style, we can think of `reset` as introducing a fresh `shift` that is only available in the *lexical scope*.
4. Effekt – A Library Design

```scala
new Collect handle { amb ⇒
  var x = 0
  amb.flip() map { b ⇒
    if (b) { x = 2 } else {}
  }
} ▶ List(2, 2)
```

(a) Example illustrating the difference between global (left) and local, backtrackable state (right). Access to the state is highlighted.

```scala
trait State {
  def Field[T](init: T): Field[T]
}

trait Field[T] {
  def get(): Control[T]
  def put(value: T): Control[Unit]
  def update(f: T ⇒ T): Control[Unit] = get().map(f).flatMap(put)
}

def region[R](prog: (s: State) ⇒ Control[R]): Control[R] = ...  
```

(b) Effect signature of the state effect and its built-in handler region.

Figure 4.7. The State effect: mutable state that interacts well with multiple resumptions.
4.3 Ambient State

Many practical handler implementations require some form of mutable state. Our host-language Scala already supports mutable state and effect handlers can readily use it. However, combining mutable state and delimited control can interact in unforeseen ways (Kiselyov et al., 2006; Leijen, 2018b). This is illustrated by the example in Figure 4.7a. In the program on the left, we introduce a local mutable variable x. Only if the flip operation returns true we modify it. Still, running the example outputs List(2, 2). Surprisingly, the change to x is also visible in the branch where flip returns false. Built-in mutable state is global and does not backtrack across different resumptions.

However, for some effect handlers, we want local, backtrackable state. Running the example program should yield List(2, 0), backtracking the local state when resuming for the second time. Leijen (2017c) calls this form of state “ambient”. There are multiple ways to obtain local, backtrackable state. As in Section 3.3.4, we could define a state effect in terms of control effects. This technique has been presented by Kammar et al. (2013) for effect handlers and by Kiselyov et al. (2006) for delimited control. Alternatively, we could also offer a generalized form of effect handlers that support local state. For example, Koka chooses this solution and supports “parameterized handlers”, which manually perform state-passing (Leijen, 2017c).

For our design of Effekt, we decided to offer a built-in State effect that exhibits the correct backtracking behavior when combined with our implementation of delimited control. Figure 4.7b defines the interface of the effect signature State and the corresponding built-in handler region. The effect signature State contains a nested type Field with the necessary operations to retrieve (i.e., get) and update (i.e., put and update) the state. The use of the state effect is illustrated in the right column of Figure 4.7a. Given a capability state we create a new field x with

```scala
val x = state.Field(...)
```

To read and update the state, we use the effect operations get and put on the field. In our design, state access is effectful and thus operations get and put return a result in the Control monad.

Our implementation of Control is specialized to properly save and restore the fields for each State effect. The region handler introduces a new mutable frame, which holds the allocated fields on the stack that the implementation of Control is based on. Capturing the continuation with shift will capture parts of the stack, shallowly copying the current values of the fields. Calling the continuation restores the values. This implements dynamically scoped state (Kiselyov et al., 2006) and allows constant time access and modification at the cost of copying on continuation capture. If we would switch the order of the effect handlers to

```scala
region { state ⇒ new Collect handle { amb ⇒ ... }}
```

running the example would again yield List(2, 2). In this order, changes to the state are persistent across multiple resumptions. In the next section, we will encounter a few examples that use this state effect.

**Note** Our design for ambient state is only necessary, since continuations in Effekt can be resumed multiple times. Continuations in Effekt are thus multi-shot (Hieb et al., 1990), while for instance continuations in Multicore OCaml (Dolan et al., 2014) are one-shot\(^\text{13}\) (Dolan et al., 2015). Restricting Effekt to one-shot continuations, it would be safe to use native mutable state, since we cannot observe the difference between local and global state anymore.

\(^\text{13}\)Multicore OCaml offers support to manually clone continuations. This way, continuations can also be resumed multiple times (Kiselyov and Sivaramakrishnan, 2018).
4. Effekt – A Library Design

4.4 Composing Effect Signatures

In the previous Section 4.2.4, we have seen how to program directly with multi-prompt delimited control. Comparing the implementations of handler functions maybe and collect with the handlers Maybe and Collect, we find that there is only a small difference between programming with multi-prompt delimited control (Section 4.2) and programming with effect handlers (Section 4.1). In the following sections, we introduce the missing interfaces to program with effect handlers.

We also give additional examples to evaluate the different dimensions of extensibility gained by embedding Effekt into Scala. Mapping effect signatures and handlers to existing features of object-oriented programming allows us to reuse the abstractions those features offer. In particular, as we will see, mapping signatures to Scala’s traits opens up interesting new modularity benefits. With the advent of default methods (Gosling et al., 2015), many of those benefits also apply to Java’s interfaces. Here, we revisit some of the abstractions and highlight the modularity benefits newly gained by implementing effect handlers this way.

We will see how to compose effect signatures, effect handlers, and effectful programs.

4.4.1 Extending Effect Signatures

Since signatures are traits (Odersky et al., 2006), we can extend them and add new operations. Here we extend the Amb trait (Section 4.1) with an additional effect operation choose.

```scala
trait Choose extends Amb {
  def choose[A](first: A, second: A): Control[A]
}
```

This introduces a subtyping relationship between Amb and Choose capabilities. A Choose handler thus can also be used to handle Amb. This cannot be expressed in Koka, for example, where a handler handles precisely the effects of a single given effect signature (Leijen, 2017c).

4.4.2 Default Methods: Primitive vs. Derived Effect Operations

Traits in Scala cannot only contain abstract method declarations, but also concrete method implementations. Similarly, our effect signatures cannot only contain abstract operations, but also concrete effect operation implementations, as illustrated in the following example.

``` scala
trait Fiber {
  // primitive effect operations
  def suspend(): Control[Unit]
  def fork(): Control[Boolean]
  def exit(): Control[Nothing]

  // derived effect operations
  def forked(p: Control[Unit]): Control[Unit] = for {
    b ← fork()
    r ← if (b) p andThen exit() else pure(())
  } yield r
}
```
4.4. Composing Effect Signatures

Here, the Fiber effect (Dolan et al., 2015) for cooperative multitasking has abstract effect operations `suspend`, `fork`, and `exit` that need to be implemented by handlers. It additionally contains a concrete effect operation `forked`, which is implemented in terms of `fork` and `exit`. We refer to operations like `suspend` as primitive effect operation and to operations like `forked` as derived effect operations. Of course, in other languages with effect handlers like Koka, `forked` could be defined as a simple effectful program using the Fiber effect. In Effekt, however, handlers that implement the Fiber effect can choose to overwrite the `forked` implementation – for instance for efficiency purposes.

The derived operation `forked` does not make any assumptions about the handler implementation. In particular, it does not explicitly capture the continuation with `shift` but only uses the other effect operations of Fiber. This illustrates an important difference to languages like Koka, where every effect operation always automatically captures the continuation (Leijen, 2017c). As a result, in Koka, the effect operation call `fork()` would not refer to the same handler, but an outer one. In our example, however, `forked` does not explicitly capture the continuation and it is hence safe to call `fork` while referring to the same handler. For similar reasons, the following implementation of `forked` could potentially result in a runtime error:

```scala
def forked[E](p: Control[Unit]) = use { resume ⇒ fork() flatMap resume } // BAD
```

In Section 6, we will see how to embed an effect system in Scala to statically reject implementations like the above.

4.4.3 Abstract Type Members: Effect Signatures as Module Interfaces

Another particularly interesting example of abstraction reuse are Scala’s abstract type members (Odersky and Zenger, 2005b). Mapping effect signatures to Scala traits, signatures cannot only describe effect operations, but also have (abstract) type members. This opens up interesting ways to structure effect signatures, illustrated by the signature for the async effect (Dolan et al., 2017):

```scala
trait Async {
  type Promise[T]
  def async[T](prog: Control[T]): Control[Promise[T]]
  def await[T](p: Promise[T]): Control[T]
}
```

The abstract type member `Promise` allows handler implementations to choose the representation of promises. The effect signature declares two effect operations (`async` and `await`) that refer to the abstract type. In a concrete capability of type `Async`, this choice of representation is hidden existentially. Odersky and Zenger (2005b) show how to use abstract type members and self-type annotations to express family polymorphism (Ernst, 2001): multiple (potentially mutually recursive) types form a family. Family polymorphism requires that the involved types can be covariantly refined together as a family, similar to open recursion on the level of types. With Effekt, the implementation strategy of using abstract type members to express family polymorphism now can also be applied to effect signatures and effect handlers. In our example, hiding the representation of promises behind an abstract type member ensures that promises can only be awaited by the capability that instantiated them.
4. Effekt – A Library Design

Example: Asynchronous Programming

Having declared the effect signatures for Async and Fiber we can write effectful programs using these effects:

```scala
def asyncExample(f: Fiber, a: Async) = for {
  p ← a.async { for {
    _ ← log("Async 1")
    _ ← f.suspend()
    _ ← log("Async 2")
    _ ← f.suspend()
  } yield 42 }
  _ ← log("Main")
  r ← a.await(p)
  _ ← log("Main with result " + r)
} yield ()
```

This example illustrates how two advanced control-flow structures, one for asynchronous programming and one for cooperative multitasking, can naturally be used together. We will see how to run this example after having defined the handlers for Fiber and Async.

4.4.4 Nested Traits: Families of Effectful Types

Abstract type members like Promise are not the only way to express a family of effectful types. Traits in Scala can also be nested, and we can refactor the effect signature Async to:

```scala
trait AsyncNested {
  trait Promise[T] { def await: Control[T] }
  def async[T](prog: Control[T]): Control[Promise[T]]
}
```

This is reminiscent of the State effect (Figure 4.7b), where the trait Field is expressed as nested type.

4.4.5 Mixing Effect Signatures

Scala supports mixin composition on traits (Odersky et al., 2006). This way, we can mix independently declared effect signatures:

```scala
trait Nondet extends Amb with Exc
```

Furthermore, since traits can contain both abstract and concrete definitions, effect signatures can be mixed to mutually implement primitive (i.e., abstract) effect operations in one signature by derived (i.e., concrete) effect operations in another. This mutually recursive matching of definitions works for both, abstract types and abstract methods (Odersky and Zenger, 2005b).
4.5 Composing Effect Handlers

In this section, we introduce the missing interfaces to program with effect handlers.

4.5.1 From Delimited Control to Effect Handlers

We purposefully presented programming with delimited control close to programming with effect handlers to highlight one small, but important difference:

“the handling of the continuation takes place at the identical site as the creation of the continuation” — Sitaram (1993, p. 148)

For the purpose of this section, we like to rephrase this quote to:

Effect handlers syntactically encapsulate the introduction of a prompt and its use.

Specifically, users do not manually pass prompts around, to then use them at arbitrary points in the program and to capture the continuation. Instead, effect operations that use a prompt are lexically related to the reset that introduced the prompt. As pointed out in Chapter 2, we believe that this is one of the most important aspects that make programming with effect handlers more approachable than programming with (multi-prompt) delimited control. Of course, we can voluntarily restrict ourselves to this mode of use, as we did in Section 4.2.4. While explicitly using reset and closing over prompts is a good way to understand the details of delimited control, it surfaces too much technical detail. In the following, we focus on effect operations and the high-level abstraction of handlers. In particular, each handler instance contains exactly one prompt marker, which it only uses internally. It would be possible to use the handler abstraction together with reset and have handlers close over prompts, as in:

```java
reset { p ⇒ new Handler(p).handle(...) }
```

However, we want to fully hide prompts as a concept from users of Effekt. To ease the instantiation of handler, we thus include the following library-internal function, which corresponds to pushPrompt by Dybvig et al. (2007).

```java
def resetWith[R](prompt: Prompt[R])(prog: Control[R]): Control[R]
```

Of course, we can express reset in terms of resetWith:

```java
def reset[R](prog: Prompt[R] ⇒ Control[R]): Control[R] = {
  val p = new Prompt[R] {}; resetWith(p)(prog(p))
}
```

We are now finally ready to fully implement the Handler interface of Figure 4.3b. The expressive power of effect handlers comes from the two operations handle and use, which are encapsulated in the library trait Handler. Figure 4.8 shows the implementation of these two operations in terms of delimited control. Handlers internally create a prompt marker of type Prompt[E] with the effect domain E. The answer type of delimited continuations thus will be the effect domain E.

In the implementation of handle, we install the prompt marker as a delimiter before resuming with prog. The handler thus delimits the extent of captured continuations. By running unit
4. Effekt – A Library Design

trait Handler[R, E] {
  protected def unit(result: R): Control[E]

  // Each capability contains a unique prompt
  private val prompt = new Prompt[E] {}

  // Handlers encapsulate capturing ...
  protected def use[A](body: (A ⇒ Control[E]) ⇒ Control[E]): Control[A] =
    shift(prompt)(body)

  // ... and delimiting the continuation in one interface
  def handle(prog: this.type ⇒ Control[R]): Control[E] =
    resetWith(prompt)(prog(this) flatMap unit)
}

Figure 4.8. The handler implementation – handlers contain a prompt marker, use captures the continuation and handle delimits the scope.

after prog, programs that use no further effects will be lifted from R to E. The method use calls shift with prompt as a prompt marker to capture the continuation up to the most recent call to handle on this very handler instance. The method unit is still left abstract, as it needs to be implemented by concrete handlers like Collect and Maybe.

Effect handlers in Effekt are deep handlers (Kammar et al., 2013). That is, all effect operations are recursively handled by the same handler. To implement deep handlers, we choose −shift + as underlying semantics of Control. This asserts that all subsequent calls to use on this handler are again delimited by prompt. Our operations handle and use are thus conceptually very similar to spawn and the corresponding controller by Hieb and Dybvig (1990).

Defining handlers as traits allows us to use mixin composition and thereby discover new opportunities for extensible handler definitions, which we explore in the remainder of this section.

4.5.2 The Effect Expression Problem

Many implementations of libraries and languages for (algebraic) effects and handlers are based on a deep embedding (Boulton et al., 1992) of effect operations (Kiselyov and Ishii, 2015). They reify effect operations as alternatives in a sum type and represent effectful computations as a command-response tree. For instance, the flip effect operation would be reified as a constructor of an algebraic data type Amb. Handlers fold over the tree of computation and use pattern matching to interpret the reified effect operations (Leijen, 2017c; Hillerström et al., 2017; Bauer and Pretnar, 2015; Kiselyov and Ishii, 2015; Kiselyov and Sivaramakrishnan, 2016). To mix programs with different effects means to extend an open union type of reified effect operations.

In contrast, by performing capability passing and representing effect signatures as traits, Effekt builds on a shallow embedding (Hudak, 1998; Carette et al., 2007) of effect operations. Instead of folding over the tree of computation, user programs directly call effect operations on the handler. In a language with mixin composition, shallow embeddings can be structured in a pleasingly extensible way (Oliveira and Cook, 2012). Thus, Effekt has a solution to the expression problem (Wadler, 1998) at its foundation, a property it shares with many other effect handler
implementations. For instance, languages like Koka (Leijen, 2014), Frank (Lindley et al., 2017), and Links (Hillerström et al., 2017) are based on row polymorphism (Gaster and Jones, 1996) and Extensible Effects (Kiselyov et al., 2013; Kiselyov and Ishii, 2015) are based on open unions (Swierstra, 2008).

Viewing the tree of computation as a recursive data type, we can define the effect expression problem as modularly and type safe being able
a. to implement new handlers for an effect operation – this corresponds to adding a new function definition over the recursive data type in the original expression problem;
b. to add new effect operations – this corresponds to adding a new variant to the recursive data type in the original expression problem.

The analogy to the expression problem (Wadler, 1998), however, is not perfect: Most descriptions of the expression problem only consider a single algebra, whereas with effect handlers we typically have more than one effect signature and the order of handling / folding over the operations affects the semantics.

4.5.3 Dimensions of Extensibility

We can relate extensibility dimensions discussed in the literature on the expression problem to the effect handler setting and show how Effekt supports them. Importantly, by embedding effect handlers into a general purpose programming language like Scala, the modularity features of the host language become available to structure effectful programs and handlers.

Adding new Handlers for an Effect

The first dimension of the effect expression problem. A central feature of every implementation of effects and handlers is the ability to define a new handler for an existing effect. Effekt supports this feature: users can define a new trait or class that implements an existing effect signature.

Adding new Operations to an Effect

The second dimension of the effect expression problem. We can distinguish between adding an operation to an existing effect signature, and adding a new effect signature. Effekt supports modular extension of effect signatures as illustrated by the example trait Choose of Section 4.4. Many other languages, like Koka, cannot extend or compose effect signatures. In those languages, it is therefore also not necessary to extend or compose handler implementations. In contrast, Effekt allows the programmer to extend handler implementations modularly.

```scala
trait CollectChoose[R] extends Collect[R] with Choose {
    b ← flip()
    } yield if (b) first else second
}
```

In this example, the handler for the extended effect signature Choose extends the existing Collect handler and only implements the missing effect operation choose. The example also illustrates that we can reuse the implementation of flip to implement choose.
4. Effekt – A Library Design

trait Poll extends Async {
  val state: State; val fiber: Fiber
  type Promise[T] = state.Field[Option[T]]
  def async[T](prog: Control[T]) = for {
    p ← pure(state.Field[Option[T]](None))
    _ ← fiber.forked { prog flatMap { r ⇒ p.put(Some(r)) } }
    yield p
  }
  def await[T](p: Promise[T]) = p.get() flatMap {
    case Some(r) ⇒ pure(r)
    case None ⇒ fiber.suspend() andThen await(p)
  }
}

Figure 4.9. Handler for the Async effect – using two effects State and Fiber.

4.5.4 Mixing Handlers – Horizontal Composition of Handlers

The description of the expression problem has seen many extensions and additional requirements. One additional requirement, described by Odersky and Zenger (2005a), is that the programmer should be able to combine independently developed extensions. For effect handlers this translates to compose two existing effect handlers. This feature might seem unnecessary in the context of effect handlers, where handler composition can already be expressed by nesting handlers (that is, by function composition). However, by using trait mixin composition to combine two handlers, the handler implementations can share implementation details like private methods, private state, and dependencies on other internally used effects. As an example, we define another handler for ambiguity that performs backtracking to compute only the first successful result:

trait Backtrack[R] extends Amb with Handler[R, Option[R]] {
  def flip() = use { resume ⇒ for {
    attempt ← resume(true)
    res ← if (attempt.isDefined) pure(attempt) else resume(false)
    } yield res }
}

We can implement the Nondet effect simply by mixing the handlers Backtrack and Maybe:

class FirstResult[R] extends Nondet with Maybe[R] with Backtrack[R]

The use of mixin composition is legal, since the two handlers assume the same effect domain. In general, handler implementations should be defined as traits in order to enable this form of composition. In languages like Java, using interfaces, a similar style of composition is possible but comes with a list of restrictions. In Dotty however, most of those restrictions have been lifted and traits can contain method implementations, fields, and receive constructor arguments. Using the composed handler FirstResult, we can handle Exc and Amb simultaneously:

val res3: Option[String] = run { new FirstResult handle { n ⇒ drunkFlip(n, n) } }
  ▶ Some("Heads")
4.5. Composing Effect Handlers

The example illustrates how handlers can be composed horizontally with mixin composition under the condition that they interpret the effects into the same effect domain. Operationally, they share the same prompt marker. By subtyping, the combined handler can be used to handle both effects. In res3, the capability is passed down twice, once to handle the Amb effect and once to handle the Exc effect. Every handler trait only implements one aspect of the overall handler component, which is then created by mixin composition. Based on traits and mixin composition, many OOP modularization strategies can now be applied to implement handlers:

- Programmers can use method overriding, super calls, and dynamic dispatch to model effectful extension points as methods that can be overridden in subclassing handler traits.

- Programmers can use abstract methods to describe the required interfaces of a handler trait and concrete method implementations to describe the provided interface. Mixin composition then matches the abstract and concrete method implementations based on the method signatures. This also allows expressing mutually recursive dependencies between handler components (Odersky and Zenger, 2005b).

- Programmers can use visibility modifiers to control how effect operations can be accessed. This way, helper effect operations can be restricted to be only locally accessible from the current handler implementation.

4.5.5 Composition over Inheritance – Vertical Composition of Handlers

Effect handlers allow us to locally handle a subset of effects, used by a program. To do so, handlers can again use effects in their implementation, which are then handled by other handlers. That is, we can compose handlers vertically.

Different to most other formulations of effect handlers, handlers in Effekt do not have to capture and use the continuation and consequently do not have to inherit from the Handler trait. It is up to the handler implementation to decide. Instead, it uses the effects State and Fiber and therefore requires the capabilities state and fiber as abstract value members. We define Promise to be type state.Field. That is, we store the result of the asynchronous computation in a field provided by the state capability. The handler function poll takes the two required capabilities to construct an instance of the handler trait Poll:

```haskell
def poll[R](s: State, f: Fiber)(prog: Async ⇒ Control[R]): Control[R] = prog(new Poll {val state = s; val fiber = f})
```

Compared to the expression problem literature, this forwarding to other handlers is reminiscent of family self references (Oliveira et al., 2013) or base algebras (Hofer et al., 2008).

4.5.6 Vertical Composition and Continuation Capture

The Poll handler (Figure 4.9) uses two effects State and Fiber in its implementation, but does not capture the continuation. All other effect handlers we have seen so far did inherit from Handler to capture the continuation but did not use any other effects. However, there is another way to use effects that we have not seen so far: First capture the continuation and then use another effect. Figure 4.10 and uses this technique and adapts the handler implementation by Dolan et al. (2015) to implement the Fiber effect as a round robin scheduler. In addition
4. Effekt – A Library Design

```scala
trait Scheduler[R] extends Fiber with Handler[R, Unit] {
  val state: State

  def unit(r: R) = pure(())

type Queue = List[Control[Unit]]
lazy val queue = state.Field[Queue](Nil)

  def exit() = use { resume ⇒ pure(())}
  def fork() = use { resume ⇒
    queue.update { resume(true) :: resume(false) :: _ } andThen run
  }
  def suspend() = use { resume ⇒
    queue.update { _ appended resume(())} andThen run
  }

  private def run: Control[Unit] = queue.get() flatMap {
    case Nil ⇒ pure(())
    case p :: rest ⇒ queue.put(rest) andThen p andThen run
  }
}
```

Figure 4.10. Handler for the Fiber effect – using the State effect after capturing the continuation.

to inheriting from Handler, it also requires the state capability. The difference now is in the combined usage of continuation capture (via use) and the state capability. The implementation of the effect operation exit is not interesting, as it simply discards the continuation. However, the two remaining operations fork and suspend both capture the continuation to then use the state within the body passed to use. As seen in Section 4.3, the interaction between state and continuation capture can be subtle. For the Scheduler handler, the state should be persisted across different fibers, forked by resuming once with true and once with false. That is, the state effect is required to be the outer handler. This also becomes visible in the handler function, which requires the state capability:

```scala
def scheduler(st: State)(prog: Fiber ⇒ Control[Unit]) =
  new Scheduler { val state = st } handle { fiber ⇒ prog(fiber) }
```

Capturing the continuation to then use some other effect can easily lead to runtime error. This is, for example, the case if the captured continuation contains the delimiter of the other effect. Using the other effect, after capturing the delimiter in the continuation results in a runtime error. In Section 6.1, we will see how statically reject those programs by embedding an effect system. We then revisit the scheduler example, using the effect-safe variant of ScalaEffekt (cf. Figure 6.6b).
Example: Running Asynchronous Programs

Having defined handlers for the Fiber and Async effects, we can finally run the program `asyncExample`, which results in the output on the right:

```
region { s ⇒
    scheduler(s) { f ⇒
        poll(s, f) { a ⇒
            asyncExample(f, a)
        }
    }
}
```

4.6 Composing Effectful Programs

By passing capabilities explicitly, in the presence of multiple instances of the same effect, we are able to explicitly select which instance of an effect to use. For example, we can use multiple instances of Fiber in one program to model different thread pools. At the same time, passing capabilities explicitly can be a burden since it introduces manual boilerplate. We illustrate this by reimplementing an example from Section 3.3: an effectful program that uses effects to express a parser (Leijen, 2016):

```
// AB ::= a AB | b

    accept('a')(in, exc) andThen parseAsB(amb, exc, in) map {_ + 1 },
    accept('b')(in, exc) map { x ⇒ 0 })(amb)

def accept(exp: Char)(in: Input, exc: Exc) = in.read() flatMap { t ⇒
    if (t == exp) pure(())
    else exc.raise("Expected " + exp) }

def alternative[A](fst: Control[A], snd: Control[A])(amb: Amb) =
    amb.flip() flatMap { b ⇒ if (b) fst else snd }

In the example, we use an Input effect that allows reading from an input stream.

trait Input extends Eff { def read(): Control[Char] }
```

Effect handlers allow us to choose between many different semantics without having to change concrete parsers, like `parseAsB`. To implement breadth-first (Swierstra, 2009) parsing, we can view alternatives in a grammar as cooperative (parsing) processes (using the Fiber effect) and use effect handlers like `Scheduler` to schedule these processes (Dolan et al., 2017). Similarly, an alternative handler implementation of Input could capture the continuation at the call to the effect operation `read()`. This way, to implement online (or streaming) parsers (Swierstra, 2009), we can then store the continuation until further input is available. The continuation of `read()` can also be seen as the derivative of the parser (Kiselyov, 2007). Offering this derivative as part of a parser combinator library to the grammar author, it is possible to express parsers for two-dimensional layout in a modular way (Brachthäuser et al., 2016).
4. Effekt – A Library Design

Composing effectful programs that use different effects, the programmer needs to manually pass the capabilities to the respective function calls. In particular, a program (like parseAsB) that uses other effectful programs (like accept and alternative) needs to take the union of all capabilities, required by its dependencies. To call them, the programmer needs to select the correct subset of capabilities and provide them along other arguments.

4.6.1 Implicits for Capability-Passing Style

While the overall design of Effekt is largely independent of Scala, there are certain features that ease the use of the library. One such feature is implicit parameters (Odersky, 2019a). Implicit parameters (called “given-clauses” or “contextual parameters” in Scala 3 (Odersky, 2019c)) can help to automatically pass function arguments based on their type (Odersky et al., 2017). This makes implicits a perfect fit for the Effekt library.

Implicit Parameters for Effectful Functions

To implicitly lookup capabilities, for every effect signature we define functions like:

```scala
def Amb given (a: Amb) = a
```

Calling the nullary function Amb implicitly searches for a value of the equally named type in the current scope and returns it. Using these helpers, we can now rewrite the above example to:

```scala
def parseAsB given Amb given Exc given Input: Control[Int] = alternative(
  accept('a') andThen parseAsB map { _ + 1 },
  accept('b') map { x => 0 })
```

```scala
def accept(exp: Char) given Input given Exc = Input.read() flatMap { t =>
  if (t == exp) pure(())
  else Exc.raise("Expected " + exp) }
```

```scala
def alternative[A](fst: Control[A], snd: Control[A]) given Amb =
  Amb.flip() flatMap { b => if (b) fst else snd }
```

The signature of accept informs us that it relies on an instance of Input and an instance of Exc being available in scope at the call site. At the same time, it brings these two instances in scope for the method body, so Input.read and Exc.raise will resolve to method calls on the corresponding implicit argument. Note that Input is the nullary method call to a boilerplate function as defined above. For the purpose of this chapter, it is enough to understand that implicit search is performed at compile time, is lexically scoped, and type directed. Implicit resolution results in a program very similar to the explicit capability passing variant above. We can also choose to bind implicit parameters to explicit names.

```scala
def parseAsB given (a: Amb) given (e: Exc) given (i: Input): Control[Int]
```

Binding capabilities to names also enables us to fall back to passing them explicitly (e.g., `accept('a') given a given e`). This is important to resolve conflicts in case of ambiguous implicits, which result in a compile time error.

---

14Since Scala 3, naming implicitly bound variables is optional (Odersky, 2019c). The signature thus roughly corresponds to `def accept(exp: Char)(implicit $1: Input, $2: Exc) in Scala 2.`
4.6. Composing Effectful Programs

4.6.2 Binding Implicit Parameters

Similarly, we can modify the signature of the handle method to accept an implicit function type, or “contextual function” \( A \Rightarrow B \) (Odersky et al., 2017). That is, a lambda that takes an implicit parameter of type \( A \) to return a \( B \):

\[
def \text{handle}(\text{prog}: \text{given this.type} \Rightarrow \text{Control}[^R]) : \text{Control}[^E]
\]

Handlers thus introduce bindings for implicit parameters. Since the parameter to the lambda is implicit, the compiler will perform automatic eta-expansion, based on the expected function type. This allows us to rewrite the handler applications to hide capability passing altogether. Assuming adapted versions of our previous examples, we can see how implicits

\[
\text{run} \{ \text{new Collect handle \{ new Maybe handle \{ drunkFlip \} \} } \}
\]

further simplify the process of rearranging the handlers:

\[
\text{run} \{ \text{new Maybe handle \{ new Collect handle \{ drunkFlip \} \} } \}
\]

Implicit parameters are thus a perfect fit for capability passing, removing most its syntactic overhead.

4.6.3 Reducing the Overhead by Composition

Another strategy to reduce the burden of passing capabilities is by composition. We can define a trait that contains the necessary capabilities as members:

\[
\text{class Parser(given (val amb: Amb, val exc: Exc, val input: Input))}
\]

Now all three methods can be refactored to only take one (potentially implicit) argument of type \( \text{Parser} \), manually projecting the fields where necessary.

Not only functions can obtain their arguments implicitly by type – the same also holds for classes and their constructor arguments. We can rewrite the composed parser as follows:

\[
\text{class Parser given (val amb: Amb, val exc: Exc, val input: Input)}
\]

Like with implicit parameters, when creating an instance of \( \text{Parser} \), the constructor arguments can be omitted and will be filled in by the Scala compiler. Using the same technique, handlers can mark the effects they are dependent on as implicit.
4. Effekt – A Library Design

4.7 Related Work and Chapter Conclusion

In this section, we discuss closely related work. In particular, we compare our approach of explicit capability passing to other implementations of effect handlers, put our implementation of Control in perspective with freer monads (Kiselyov and Ishii, 2015), and relate our effect handlers design for object-oriented languages to others.

4.7.1 Capability-Passing Style

We start by discussing some properties of explicit capability-passing style.

Shallow embedding of effect operations  Many implementations of libraries and languages for effect handlers are based on a deep embedding (Boulton et al., 1992) of effect operations. In contrast, by performing capability passing, Effekt builds on a shallow embedding (Hudak, 1998; Carette et al., 2007) of effect operations. Interpretation of effect operations is moved from (external) pattern matching to (internal) dynamic dispatch, which makes the shallow embedding of effect operations a good fit for object-oriented programming languages. Similarly, Kammar et al. (2013) base their library implementation of algebraic effect handlers on Haskell type classes, effectively performing a shallow embedding. Using type classes and the associated dictionary passing helps Kammar et al. (2013) to achieve good performance results since it prevents the materialization of constructors for effect operations. It also avoids any search for the matching handler implementation in some kind of handler stack, as it is done in Koka (Leijen, 2017c, 2017b).

Simplified typing  Another advantage of capability passing is that it simplifies typing. Combining object-oriented programming with effect handlers, we define use as a method on Handler. As a method, it naturally shares the type of the effect domain with its implementing class. We thus use dynamic dispatch instead of implementing a pattern-matching interpreter. This helps us to avoid advanced dispatch features, such as type constructor polymorphism (Kiselyov et al., 2013) or generalized algebraic data types typically associated with interpreter based solutions (Kiselyov and Ishii, 2015). In a previous presentation of Effekt (Brachthäuser and Schuster, 2017), we represented capabilities as a pair of a prompt marker and the handler implementation. To hide answer types existentially in the user program, this required us to use type members and path-dependent types. In contrast, Effekt as presented in this chapter is designed to remove requirements on the type system and to blend in with OOP paradigms. We immediately represent capabilities as instances of type Exc and hide implementation details like answer types by means of simple subtyping.

Manually selecting effects  By performing capability passing, we require the user to explicitly select the handler to use. In other libraries and languages for effect handlers, an effect operation always resolves to the dynamically closest handler implementation. Explicitly calling methods on handlers has the advantage that no confusion arises when multiple handlers for the same effect are present. Other languages added features post hoc in order to allow programmers to select effects more explicitly. Leijen (2018b) added inject to the Koka language, and Convent et al. (2020) added mask and adaptors to Frank. Both inject and mask are conceptually similar to the lift operation proposed by Biernacki et al. (2017), which allows to select effect handlers in terms of the distance to the effect operation. For example, in Koka, flip() will be handled
4.7. Related Work and Chapter Conclusion

by the closest handler for ambiguity, whereas \texttt{inject<amb>(}\texttt{fun\{flip()\}}\texttt{)} will be handled by
the second closest handler, and so forth. This is reminiscent of programming with De Bruijn
indices (de Bruijn, 1972), where variables also refer to their binders by distance. In contrast,
with explicit capability passing different effects can be referred to by name.

\textbf{Effect polymorphism} Since capabilities are first-class, functions (and objects) can close over
them. This allows users to express some form of effect polymorphism (Osvald et al., 2016). Take
the following effectful map function, adapted from Rytz et al. (2012), as an example:

\begin{verbatim}
def mapM[A, B](lst: List[A], f: A ⇒ Control[B]): Control[List[B]]
\end{verbatim}

The signature of \texttt{mapM} only informs us that \texttt{f} has control effects, but is silent on the concrete
effects. Still, we can invoke map passing a lambda that uses an effect like \texttt{Amb}.

\begin{verbatim}
new Collect handle { amb ⇒ map(List(1,2,3), a ⇒ amb.flip()) }
\end{verbatim}

The function passed as argument \texttt{f} simply closes over the capability. The use of the \texttt{Amb} effect
does not become visible in its type, which is \texttt{Int ⇒ Control[Boolean]}. The function \texttt{mapM} is
thus effect polymorphic.

\textbf{Effect safety} Passing capabilities as evidence is not new. Osvald et al. (2016) also perform
capability passing in Scala. They present capability passing as an alternative approach to
traditional type and effect systems. A capability serves as a constructive proof, that the holder
is entitled to use the actions associated with the capability. To statically prevent the leaking of
capabilities, Osvald et al. introduce a new type system feature: arguments to functions can be
marked as second class (or “local”). Similar to region based memory management (Kiselyov and
Shan, 2008), the type checker then guarantees that the capability cannot leave the dynamic
scope of the function call. They show how second-class values can be used to model checked
exceptions, as in the following example, adopted from Osvald et al. (2016):

\begin{verbatim}
def Try[T](fn: (@local (Exception ⇒ Nothing)) ⇒ T): Option[T]
Try { throw ⇒ throw(new Exception) } // Safe usage of 'throw'
\end{verbatim}

The function \texttt{throw} is introduced by the exception handler and marked as \texttt{@local}. This guarantees
that \texttt{throw} cannot escape the dynamic scope of the function \texttt{fn}. Much like effect handlers, \texttt{try}
thus introduces a capability. Liu (2016) presents a different approach to capability based effect
safety, by distinguishing between functions that can capture capabilities and others that cannot
(called “stoic”). In our design of \texttt{Effekt}, we adopt the capability passing of Osvald et al. (2016)
and use it together with a monad for delimited control to generalize exception handlers to effect
handlers. While second-class values might be an interesting way to achieve effect safety, in
Chapter 6 we explore a different approach based on path-dependent types and intersection types.

\textbf{Performance} To implement important performance optimizations, Leijen (2017b) explicitly
tags each effect operation in a handler with information about how the continuation is used.
Similarly explicit, in \texttt{Effekt}, handlers capture the continuation with \texttt{use}. \textit{Tail resumptive}
handlers (Leijen, 2017b), \textit{i.e.} handlers that only call the continuation in tail position, do not need
to capture the continuation and do not call \texttt{use}. Not having to capture the continuation is
only possible due to our combination of capability-passing style and multi-prompt delimited
continuations. As already mentioned in Section 2.3 and what is the essence of ambient functions (Chapter 3), capturing the continuation also leads to a “shift in perspective”. That is, effects are dynamically resolved at the handler site – as opposed to at the call site of the effect operation. In Effekt, we can avoid capturing the continuation altogether since we replace dynamic lookup of effect handlers by explicit capability passing. Effect operations explicitly refer to capabilities at handler site by closing over them.

Building on the insights of this thesis, in related work (Schuster and Brachthäuser, 2018; Schuster et al., 2020), we started to explore strategies to pass capabilities at compile time. This way, programs are always fully specialized to the handlers under which they are executed.

Deep handlers and shallow embeddings Due to our design decision of a shallow embedding of effect operations, handlers in Effekt are deep handlers (Kammar et al., 2013). That is, all effect operations in the continuation captured by `use` will automatically be handled recursively by the very same handler. However, if a shallow handler semantics is required, it can be achieved by reifying the command-response trees of selected effect operations. A reifying effect handler interprets a program of type `R` into a free-structure `Frees<R>` (Kiselyov et al., 2013), which then can be interpreted step-by-step. While this encoding is possible, it can lead to performance problems and memory leaks. Hillerström and Lindley (2018) present an encoding of shallow handlers in terms of deep handlers that does not suffer from these problems.

4.7.2 Implementing Monadic Delimited Control

Kiselyov and Sivaramakrishnan (2016) embed effect handlers into OCaml. They also build on an implementation of multi-prompt delimited control (Kiselyov, 2012), which makes use of the fact that OCaml includes bytecode instructions that support manipulation of the runtime stack. Just like Kiselyov (2012), our implementation of delimited control presented in this chapter is
4.7. Related Work and Chapter Conclusion

based on the work by Dybvig et al. (2007). Since the JVM does not allow for stack manipulation, however, we build on their monad \( \mathbb{C} \) for multi-prompt delimited control (Dybvig et al., 2007). In particular, we choose to represent “continuations as sequences of frames” (Dybvig et al., 2007, Section 7.3) – but make the following adjustments:

- Dybvig et al. choose the semantics \( - \text{shift}_- \) for their control operator \texttt{withSubCont}. We instead choose \( - \text{shift}_+ \) as it is closer to deep effect handlers (Kammar et al., 2013) and allows for a more uniform representation of stack segments, which are always delimited by a prompt.

- Dybvig et al. (Section 7.3) represent continuations as a type-aligned sequence of frames. As the authors remark, capturing the delimited continuation and searching for the prompt is thus linear in the number of individual frames. We slightly refine the representation and additionally group frames, which are not separated by prompts in a nested list (Figure 4.11). Our continuation is thus a prompt separated list of frames. This representation and the choice of \( - \text{shift}_+ \) allow us to guarantee that prompt search, continuation capture, and resumption is always at most linear in the number of prompts.

- Dybvig et al. index the \( \mathbb{C} \) monad by the final answer type and then use rank-2 types to prevent mixing prompts between different “runs”. We choose to reduce the required type system features at the cost of additional unsafety. However, much like our capabilities, which can escape, Dybvig et al. (2007) also do not guarantee that every continuation capture for a particular prompt will succeed. The continuation might not be delimited, which results in a runtime error that the prompt cannot be found. Considering this, mixing prompts between different runs seems like a minor additional unsafety compared to this existing source of runtime errors. In Chapter 6, we develop an effect-safe variant of multi-prompt delimited control that rules out both sources of unsafety.

- Dybvig et al. eventually perform all computation in a monad \( \mathcal{P} \), which is essentially a state monad used to generate fresh prompts. Since the setting of our embedding is the JVM, which does not support tail call elimination, but mutable state, we use object identity as the source for prompts and instead instantiate \( \mathcal{P} \) as a trampoline (Ganz et al., 1999).

4.7.3 Effect Handlers and Object-Oriented Programming

As highlighted in Section 4.4, effect safe programming with effect handlers in a language with objects comes with new opportunities for modularization. As we will see in Chapter 6, it also comes with new challenges – mediating encapsulation and flexible use of objects. Not much prior work exists that combines effect handlers and object-oriented programming.

\texttt{Eff} (Torreborre, 2016) is a translation of the work by Kiselyov and Ishii (2015) to Scala. It almost exclusively targets the functional aspects of Scala and thus integrates well with other popular libraries for functional programming in Scala.

A notable exception is \texttt{JEff} (Inostroza and van der Storm, 2018), which appeared after our first presentation of \texttt{Effekt} and simultaneously with our publication on \texttt{JavaEffekt} (Brachthäuser et al., 2018). Inostroza and van der Storm (2018) also combine effect handlers and object-orientated programming in their language \texttt{JEff}. Similar to \texttt{Effekt}, \texttt{JEff} maps effect signatures to interfaces and handles to classes implementing the interfaces. The following example, adapted from Inostroza and van der Storm (2018), gives the effect signature for an exception effect and the corresponding handler.
4. Effekt – A Library Design

```java
interface Exc { eff Nothing raise(String s) }
class Maybe<T>() implements Exc, Handler<Option<T>, T> {
    Option<T> return(T t) = new Some(t)
    eff Nothing raise(String s) = new None()
}
```

We take the striking similarities to Effekt as empirical support for our design. Despite the similarities in the combination of object-oriented programming and effect handlers, there are also a few notable differences. JEff, as a calculus, performs dynamic lookup and continuation capture similar to $\lambda_{dch}$. In contrast, as a library embedding in Scala, in Effekt we perform capability passing instead of dynamic binding and build on a monadic implementation of delimited control. In JEff, the continuation takes an updated copy of the effect handler as additional argument. This allows both to model stateful handlers and even to change the handler implementation for the rest of the computation, similar to shallow handlers. In Effekt, we added special support for ambient state as a separate effect and handlers are deep. The design of Effekt, presented in this chapter is not effect-safe. In contrast, Inostroza and van der Storm (2018) describe an effect system for their calculus. Their effect system does not feature effect polymorphism, which rules out many important examples. In Chapter 6, we present an effect-safe variant of ScalaEffekt that also supports effect polymorphism.

4.7.4 Conclusion

In this chapter, we presented the design of Effekt, a library integrating effect handlers with object-oriented programming. We followed the mantra that programming with effect handlers in Effekt is object-oriented programming and mapped effect signatures to interfaces and handlers to classes. We highlighted several dimensions of extensibility, which are supported by Effekt and showed modularization opportunities gained by combining effect handlers with object-oriented programming. We explored capability passing as an alternative to the traditional dynamic binding of effect handlers (cf. Chapter 3). While capability passing imposes some syntactic overhead on the user, it offers new interesting perspectives on performance optimizations (such as tail resumptions) and effect polymorphism. Most of the syntactic overhead associated with capability passing can be reduced by using Scala’s implicits.

Despite the benefits, our implementation of Effekt in Scala suffers from two drawbacks: user programs have to be written in monadic style and capabilities can leave the dynamic scope of the handler. We separately address the two issues in the following two chapters. Chapter 5 presents JavaEffekt, a direct-style implementation of effect handlers for Java. Chapter 6 reiterates the design of ScalaEffekt to add an effect system that statically prevents the usage leaked capabilities.
Chapter 5

Java Effekt – Effectful Programming in Direct Style

In this chapter, we further evaluate our library design of Effekt by implementing it in another object-oriented language: Java.

The implementation of ScalaEffekt is based on a monad for delimited control. While Scala offers for-comprehensions to facilitate writing monadic code, there is no such support in Java. Writing monadic code in Java is non-idiomatic and verbose. In this chapter, we overcome this limitation and allow user programs to be written in direct style. To achieve this, we present a framework that consists of three core components: A type selective continuation-passing-style (CPS) transformation on the level of Java Virtual Machine (JVM) bytecode, an implementation of delimited continuations on top of the bytecode transformation, and finally a library for effect handlers in terms of delimited continuations.

We evaluate and compare the performance of our bytecode-transformation-based implementation against other bytecode transformations that allow continuation capture. The measurements indicate that the performance of our library is competitive with other existing transformations.

The programming language Java lacks a general mechanism to express advanced control flow. In consequence, many control-flow abstractions like generators\(^\dagger\), asynchronous programming with async/await\(^\ddagger\), the coroutine programming model\(^\S\), and fibers\(^\S\), that is lightweight user-level threads, are currently implemented by custom source-to-source or bytecode transformations. Since each extension makes different assumptions about the generated code, combining the different concepts in a single project ranges from non-trivial to impossible. As has been shown in the literature, effect handlers can express many of these control-flow abstractions as simple libraries (Dolan et al., 2015, 2017; Leijen, 2017a). Having support for effect handlers thus would naturally allow a combined usage of those features. We will revisit some examples of those control-flow abstractions in the context of Java in Section 5.3.

In this chapter, we present an implementation of the Effekt design in Java, which we call JavaEffekt. As in the previous chapter, we follow our mantra that programming with effect handlers in JavaEffekt is object-oriented programming. Hence, effect signatures are Java interfaces, effect

---

\(^\dagger\) https://github.com/peichhorn/lombok-pg/wiki/Yield
\(^\ddagger\) https://github.com/electronicarts/ea-async
\(^\S\) https://github.com/offbynull/coroutines
\(^\S\) https://github.com/puniverse/quasar

5. Java Effekt – Effectful Programming in Direct Style

![Diagram of the Java Effekt framework]

**Figure 5.1.** Structure of the JavaEffekt framework. Directed, solid arrows express dependencies.

handlers are Java classes that implement those interfaces, capabilities are instances of handlers, and effectful functions are Java functions that use capabilities.

In contrast to the previous chapter, user programs in this chapter can be written in direct-style. Importantly, this way programmers can use effects and handlers together with existing control-flow mechanisms like branching (i.e., `if` and `switch`), loops (i.e., `while`), and exceptions. Also, in contrast to our implementation in Scala, in JavaEffekt, local variables (but not heap-allocated fields) show the correct backtracking behavior. To enable this, our implementation consists of three components: A type-selective CPS transformation of bytecode, an implementation of delimited continuations on top of the bytecode transformation, and a library for effect handlers in terms of delimited continuations. While all three components are designed in concert to implement the effect handler library, they can be used and understood individually. The bytecode transformation is performed independent of Java as the source language and could potentially be reused with other JVM languages such as Scala, Kotlin, JRuby, Clojure, and others.

In short, the contributions of this chapter are:

- The first library design for programming with effect handlers in Java.
- An implementation of multi-prompt delimited continuations in Java. It uses trampolining and avoids the typical linear overhead of restoring the stack upon resumption common to all continuation libraries in Java that we are aware of.
- A type-selective, signature preserving CPS transformation of JVM bytecode. We use closures introduced in Java 1.8 (Gosling et al., 2015) to create specialized instances of continuation frames. The general idea is applicable to any VM-based language that supports closure creation.
- A performance evaluation, comparing JavaEffekt with other libraries that perform bytecode instrumentation to allow continuation capture. The measurements confirm the asymptotic improvement that we obtain by trampolining. We also compare the performance of JavaEffekt with our implementation ScalaEffekt and another monadic library in Scala.

**Overview**

The chapter structure follows the structure of our framework which is presented in Figure 5.1.

**User code** We show how to program with effect handlers in Java using our library. Section 5.1 briefly presents the library design of JavaEffekt by means of our standard running example; Section 5.3 illustrates the expressiveness of JavaEffekt with several more complex examples.

102
5.1 Programming with Effect Handlers in JavaEffekt

The design of JavaEffekt follows the design principles of Effekt. In this Section, we briefly review our running example to highlight similarities and point out differences.

Figure 5.2a uses JavaEffekt to express our running example. As before, the two effect operations flip and raise are declared in corresponding effect signatures Amb (for ambiguity) and Exc (for exceptions) in Figure 5.2b. The biggest difference compared to the implementation in Scala (Chapter 4) is that the program now is written in direct style. This is also visible in the type signature of drunkFlip: Previously, we used the monadic type constructor Control to signal that the program might use control effects and used for-comprehensions to sequence effectful programs. In contrast, effectful programs now are declared as such by adding the
checked exception `Effects` to their `throws` clause. Like other bytecode instrumentation such as “Quasar” (Parallel Universe Software Co., 2013), we use checked exceptions as a coarse grained effect system (Section 5.5 provides a comparison of `JavaEffekt` with Quasar). This allows us to distinguish pure programs (that do not use control-effects) from effectful programs. This information is important to guide our bytecode instrumentation. For our CPS translation to be sound, the `Effects` exception should never be handled in user code.

5.1.1 Handling Effects

Analogous to the previous chapter, we first review how effects are handled before we give the full implementation of the involved handlers.

```java
class Maybe<R> extends Handler<R, Optional<R>> implements Exc {
    ...}
class Collect<R> extends Handler<R, List<R>> implements Amb {
    ...}
```

As before, the handlers extend our library interface `Handler<R, E>` to express that they represent effect handlers. In `JavaEffekt`, using the handlers to handle effects `Amb` and `Exc` looks like:

```java
List<Optional<String>> res1 =
    Handler.handle(new Collect<Optional<String>>(()), amb →
        Handler.handle(new Maybe<String>(), exc →
            drunkFlip(amb, exc)));  
```

Due to the lack of singleton types in Java, the method `handle(h, body)` is now a static method on `Handler`. Compared to the Scala implementation of `Effekt`, this is the only superficial difference. Running `drunkFlip` with both effects handled yields for `res1`:

```
▶ [Optional["Heads"], Optional["Tails"], Optional.empty]
```

5.1.2 Implementing Effect Handlers

We will now revisit how handlers are implemented. Figure 5.3b shows the Handler interface, which is relevant for implementing effect handlers. It also shows the type of effectful functions `Eff<S, T>`, which is just like the Java function interface `Function<S, T>`, but with its single abstract method being marked as throwing `Effects`. The interface `Eff<S, T>` is thus the Java equivalent of a function `S ⇒ Control[T]` in Scala. Similarly, interface `CPS<A, E>` corresponds to the nested effectful function type `Eff<Eff<A, E>, E>` (that is, `(A ⇒ Control[E]) ⇒ Control[E]).

Like in our Scala implementation, both handlers `Maybe` and `Collect` implement their effect operations in terms of `use` (Figure 5.3a). To implement the `raise` effect, the handler `Maybe` captures the continuation and deliberately discards it. The handler for ambiguity, in turn, captures the continuation and invokes it twice, each time yielding a list of possible results. Specialized to the `Collect` handler, the type of the captured continuation is `Eff<Boolean, List<String>>`. Hence, it is safe to finally concatenate the two resulting lists. Similarly, in `raise` it is safe to discard the continuation and immediately return `Optional.empty` because the caller of `handle(new Maybe<String, Optional<String>>(()), ...)` expects a value of type `Optional<String>`.

---

20 `Optional<A>` is an interface for optional values of type `A`, introduced in Java 1.8.
21 For this example, we assume lists to be immutable and only be constructed by `singleton` and `concat`. 

104
5.1. Programming with Effect Handlers in JavaEffekt

class Maybe<R> extends Handler<R, Optional<R>> implements Exc {
    Optional<R> unit(R r) { return Optional.of(r); }
    <A> A raise(String msg) throws Effects {
        return use(k → Optional.empty());
    }
}

class Collect<R> extends Handler<R, List<R>> implements Amb {
    List<R> unit(R r) { return Lists.singleton(r); }
    boolean flip() throws Effects {
        return use(k → Lists.concat(k.resume(true), k.resume(false)));
    }
}

(a) The two effect handlers Maybe and Collect utilizing use to capture the continuation.

abstract class Handler<R, E> {
    E unit(R r) throws Effects;
    <A> A use(CPS<A, E> body) throws Effects { ... }
    static <R, E, H extends Handler<R, E>> E handle(H h, Eff<H, R> p) throws Effects {
        ...
    }
}

interface Eff<S, T> { T resume(S value) throws Effects; }
interface CPS<A, E> { E apply(Eff<A, E> k) throws Effects; }

(b) Interface of the library class Handler and the necessary functional interfaces.

Figure 5.3. Implementation of effect handlers for Exc and Amb using the library class Handler.
5. Java Effekt – Effectful Programming in Direct Style

5.2 Implementing Effect Handlers for Java in three Steps

As can be seen from our running example, programming with effect handlers in JavaEffekt is almost just standard Java programming. Only the control operator use and its counterpart handle make the difference in expressivity. This section describes how these control operators can be implemented, bottom up. We start with a CPS translation, that rewrites all methods annotated with throws Effects, build a library for delimited continuations upon the translation, and finally implement effect handlers in terms of delimited continuations.

5.2.1 Step 1: Type Selective CPS Transformation by Example

To support accessing the continuation with use, the JavaEffekt framework performs a type selective CPS transformation by instrumenting (that is, rewriting) JVM bytecode. This can either be achieved by hooking into the class loading mechanisms of Java and transforming a classfile at runtime when it is loaded, or “ahead of time” by a separate pre-processing phase that rewrites the class files once (Binder et al., 2007); our implementation supports both. Implementing the transformation on the level of JVM bytecode opens up the opportunity of reuse for other JVM languages. While the implementation of JavaEffekt rewrites JVM bytecode, for easier accessibility this section presents the CPS transformation as a semantically equivalent source-to-source rewriting of the example program drunkFlip. This section provides an overview, Section 5.4 formally describes the bytecode transformation and explains how we treat control flow and exceptions.

Effect calls and entrypoints Figures 5.4b and 5.4c show the result of transforming the method drunkFlip. We instrument only effectful methods and identify those by means of the checked exception Effects. Using Reynolds (1972) terminology we only consider methods marked with throws Effects to be “serious”. All other functions are “trivial” and do not require any instrumentation. Consequently, we also only instrument call sites of effectful methods (effect calls). In drunkFlip, there are three such effect calls, two to flip and one to raise. We exclude tail effect calls from the translation, that is, effect calls immediately followed by a return. For drunkFlip, this means that we instrument the two flip calls, since the call to raise is in tail position. We call the code immediately following an effect call an entrypoint. We also treat the initial entrypoint of a function as an entrypoint in this sense. Similar to Prokopec and Liu (2018), for each entrypoint in an effectful method, we generate a separate entrypoint method. For our example, these are the methods drunkFlip0 (the initial entrypoint method), drunkFlip1 and drunkFlip2 (corresponding to the two invocations of flip). Importantly, entrypoint methods take the function local state as arguments. That is, all values on the operand stack and all local variables that are needed to resume the function execution after the effect call would return.

The stack interface and continuation frames Similar to how the JVM pushes a stack frame before it enters a method call (Lindholm et al., 2015), we rewrite every effect call to push a continuation frame. By calling Effekt.push, the continuation frame is pushed to a global, user-level stack. Class Effekt has a global, static field Effekt.stack that implements the interface Stack shown in Figure 5.5. The interface Stack contains all necessary methods used by the instrumented

---

22For the example in this section, we manually verified that the bytecode of the source-to-source transformation is equivalent to the result of the bytecode transformation (modulo some superfluous register stores/loads).
5.2. Implementing Effect Handlers for Java in three Steps

```java
String drunkFlip(Amb amb, Exc exc) throws Effects {
    boolean caught = amb.flip();
    if (!caught) {
        return exc.raise("We dropped the coin.");
    } else {
        return amb.flip() ? "Heads" : "Tails";
    }
}
```

(a) Source method with highlighted effect calls.

```java
String drunkFlip(Amb amb, Exc exc) throws Effects {
    Effekt.push(() → drunkFlip0(amb, exc));
    return null;
}
```

(b) Method stub, only pushing the initial entrypoint.

```java
static void drunkFlip0(Amb amb, Exc exc) {
    boolean caught = amb.flip();
    if (!caught) {
        exc.raise("We dropped the coin.");
    } else {
        bool res1 = Effekt.result();
        Effekt.returnWith(res1 ? "Heads": "Tails");
    }
}
```

(c) Entrypoints as separate, static methods.

**Figure 5.4.** CPS translation of the example in Figure 5.4a, presented as a source-to-source transformation.

bytecode. We assume that all methods on Stack are available as static methods on Effekt and write Effekt.push instead of Effekt.stack.push. In the presence of multiple threads, we would store the current stack in a thread local variable, such that each thread maintains its own stack, but refrain from doing so for simplicity. A continuation frame is an instance of the Frame interface, also shown in Figure 5.5. We use Java 8 lambdas (Gosling et al., 2015) to create instances of the Frame interface. The lambdas close over the function local state. When invoked with enter, they pass the state to the entrypoint methods. Conceptually, we represent continuation frames as instances of classes that have one field for each local variable they store. After pushing the continuation frame, the entrypoint methods call the effectful method and immediately return. Thus, all effect calls become tail calls in the translated program.

**Custom calling convention** Instrumented effectful methods use a special calling convention: We rewrite all returns to Effekt.returnValue calls. Correspondingly, entrypoint methods use Effekt.result to obtain the result of the previous effect call. We transform the original method drunkFlip to a stub (Figure 5.4b) that pushes a continuation frame for the initial entrypoint and immediately returns a dummy value. Callers of drunkFlip have to use Effekt.result to eventually get the actual return value. Our entrypoints follow the same calling convention for consistency, as we can observe in drunkFlip2. The calling convention is motivated by the goal to maximize interoperability with other Java features with as little specialization of the transformation as possible. The most important consequence of this design goal is that our translation preserves method signatures. It thus does only translate terms, not types or signatures. A standard CPS translation changes the type of a computation that returns A to a function type (A → R) → R for some answer type R (Meyer and Wand, 1985). However, this
interface Stack {
    // special calling convention
    void returnWith(Object r);
    void unwindWith(Throwable t);
    <A> A result() throws Throwable;

    // stack of frames
    void push(Frame frame);
    Frame pop();
    boolean isEmpty();
    void trampoline();
}

interface Frame {
    void enter();
}

abstract class RTStack implements Stack {
    Object res; Throwable exc;
    void returnWith(Object r) { res = r; exc = null; }
    void unwindWith(Throwable t) { res = null; exc = t; }
    <A> A result() throws Throwable {
        if (exc != null)
            throw exc;
        return (A) res;
    }
    void trampoline() {
        while (!isEmpty())
            try {
                pop().enter();
            } catch (Throwable t) { unwindWith(t); }
    }
}

Figure 5.5. Interface and example implementation of the user-level stack. Implementations of stack operations (push, pop and isEmpty) are left abstract.

change is precluded by our decision of not changing method signatures. Instead, the continuation
\( A \rightarrow R \) is obtained via the global instance of Stack, as we will see in the next subsection. Still,
the return type of an effectful computation changes from \( A \) to \( R \). To accommodate for the change
in return type, we make effectful methods use our own custom calling convention.

Implementing the stack interface Instead of using the JVM stack for continuation frames, we
use a separate user-level stack. A canonical implementation of Stack recovering the expected
runtime behavior of the JVM stack is sketched as class RTStack in Figure 5.5. Only the methods
implementing our calling convention are provided. The straightforward stack operations push,
pop, and isEmpty are left abstract. Our stack implementation performs trampolining (Ganz
et al., 1999). To run an instrumented program\(^{23}\), we first invoke it to push a frame that
corresponds to its initial entrypoint onto Effekt.stack:

\[
\text{static } \langle A \rangle \ \text{A run}(\text{Eff<Void, A>} \ prog) \{ \\
\text{prog.resume();} \\
\text{Effekt.trampoline();} \\
\text{return Effekt.result();} \\
\}
\]

To actually start execution, we call Effekt.trampoline, which will continue to pop and enter
frames until the stack is empty. If an exception is raised, the trampoline will unwind the
user-level stack frame by frame. Each frame starts with a call to result, re-raising the exception
after restoring the method state. Section 5.4 shows more details on how we deal with exceptions.

\(^{23}\)When effectful programs do not require an argument we will use Eff<Void, A>. To avoid materializing instances
of Void and binding them, we write f.resume() as a short hand for f.resume(null) and ({} \rightarrow \ldots) instead of
(\text{unusedVoid} \rightarrow \ldots). Effectful programs of type Eff<Void, A> corresponds to values of type Control[A] in our
Scala implementation.
5.2. Implementing Effect Handlers for Java in three Steps

```java
interface Prompt<E> {}
(a) Interface Prompt, used to mark positions on the stack.

class SeqStack extends RTStack {
    Seq<Frame> s = Seq.empty();
    void push(Frame frame) { s = s.push(frame); }
    Frame pop() { Frame f = s.head(); s = s.tail(); return f; }
    boolean isEmpty() { return s.isEmpty(); }
}
(b) Implementation of Stack, forwarding to an immutable stack Seq.

class DelimCC extends SeqStack implements DelimitedControl {
    <E> E resetWith(Prompt<E> p, Eff<Void, E> prog)
        throws Effects {
        s = s.mark(p); return prog.resume();
    }
    <A, E> A shift(Prompt<E> p, CPS<A, E> body)
        throws Effects {
        Seq<Frame> init = s.before(p); s = s.after(p);
        Eff<A, E> k = (A value) -> {
            s = init.prependTo(s); return (E) value; }
        return (A) body.apply(k);
    }
(c) Implementation of delimited control in terms of Seq.

interface Seq<A> {
    boolean isEmpty();
    A head();
    Seq<A> tail();
    Seq<A> before(PromptEx p);
    Seq<A> after(PromptEx p);
    Seq<A> push(A element);
    Seq<A> mark(PromptEx p);
    Seq<A> prependTo(Seq<A> init);
    static <A> Seq<A> empty() {...}
}
(d) Immutable, splittable stack.
```

Figure 5.6. Implementation of control operators. Usage of the splittable stack implementation Seq is highlighted.

5.2.2 Step 2: Delimited Continuations

A program that has been transformed with our CPS translation still uses the JVM stack for non-effectful calls but uses our user-level stack for effectful method calls. Different implementations of the interface Stack give rise to different additional operations that exploit the corresponding stack representation. In consequence, effectful programs that are executed against a particular stack implementation can make use of those additional operations. In this section, we will develop one particular implementation of Stack that implements additional operations to capture delimited continuations. All code in the rest of this chapter is subject to the CPS bytecode transformation and all methods annotated with throws Effects will be instrumented.

The Interface of Delimited Control

As already shown in Section 4.2, the effect handler library can be implemented as a very thin layer on top of multi-prompt delimited continuations. Again, our implementation of multi-prompt delimited continuations is similar to the one presented in Chapter 4. However, we translate it to Java and adapt it to our setting of bytecode instrumentation. We extend the above Stack implementation with the following interface DelimitedControl.
Instances of Prompt<E> (Figure 5.6a) are used to mark positions on the stack. The type of an effectful function Eff and of effectful programs that use an effectful continuation CPS, have been defined in Figure 5.3b. The type parameter E of Prompt unifies with the type of the computation that we delimit with resetWith. Capturing a continuation with shift, the return type of the continuation and of the body have to match the type of the prompt E. The implementation of reset in terms of resetWith directly follows the one from Section 4.4.

Using Delimited Continuations

Assuming the global stack instance supports the methods from DelimitedControl, we can reimplement the examples of Section 4.2, this time in direct style.

\[
1 + \text{Effekt.reset}(p_1 \rightarrow 10 \rightarrow \text{Effekt.shift}(p_1, k \rightarrow k.resume(k.resume(100))));
\]

\[
\downarrow 1 + (10 + (10 + 100)) = 121
\]

The second example illustrates the use of multiple prompts.

\[
2 * \text{Effekt.reset}(p_1 \rightarrow \text{Effekt.reset}(p_2 \rightarrow \text{Effekt.shift}(p_1, k \rightarrow 21)) ? 1 : 2);
\]

\[
\downarrow 2 * 21 = 42
\]

The captured continuation k contains the program segment marked by prompt p₁. We discard the continuation and replace it by the value 21.

A Splittable Stack Implementation

Figure 5.6b implements Stack by storing a list of frames in a field s. It implements all abstract operations of Stack by forwarding to this list s. For now, let’s assume Seq is an immutable implementation of a stack data structure with elements of type A. While very simple, running a program with this stack implementation already has the benefit that it performs trampolining and thus reduces JVM stack usage, which might avoid stack overflows. However, the real power of the translation comes from the fact that Stack implementations can add new methods, which expose additional (control) operators. Following Dybív et al. (2007), to implement the additional control operators resetWith and shift, we need to mark positions on the runtime stack (mark), slice the stack at given positions (before, after), and prepend whole stack segments (prependTo). The stack data structure Seq (Figure 5.6d) that we already used above offers exactly these operations. As presented in Figure 4.11, Seq<A> is a prompt-separated cons-list of elements of type A. Calling s.before(p) returns the initial segment up to (and including) the first occurrence of the marker p. This segment contains all the recently pushed elements (frames and prompts) after p has been pushed. Calling s.after(p) returns the remainder of the stack. This segment contains all elements, which have been pushed before marker p has been pushed.

As mentioned before, effect handlers in Effekt are deep handlers (Kammar et al., 2013) and our implementation of delimited control has semantics +shift+. Consequently, the stack segment obtained by seq.before(p) is delimited by the prompt marker p, while seq.after(p) is not.
5.2. Implementing Effect Handlers for Java in three Steps

Pushing a Prompt

Figure 5.6c shows the implementation of the DelimitedControl interface, using Seq. To implement resetWith, we mark the stack using the provided prompt \( p \) and update the mutable reference \( s \) with the now marked stack. We then resume with the effectful program \( \text{prog} \). Being effectful, the program \( \text{prog} \) pushes additional frames onto the stack. We can capture those frames later by slicing the stack at the position of the installed marker \( p \).

Capturing a Continuation

The control operator \( \text{shift}(p, \, \text{body}) \) captures the continuation \( k \) up to the next dynamically enclosing \( \text{resetWith}(p, \, \ldots) \) and conceptually replaces the call to \( \text{resetWith} \) with a call to \( \text{body}.\text{apply}(k) \). Its implementation in Figure 5.6c stores all frames that have been pushed after \( p \) in a local variable \( \text{init} \). This segment corresponds to the delimited continuation from type \( A \) to type \( E \). That is, the top most frame expects \( \text{Effekt.result} \) to return a value of type \( A \). The initial segment and the prompt marker are then removed from the stack by mutating it with \( s = s.\text{after}(p) \). This leaves a segment on the stack, which expects a value of type \( E \) to continue program execution. The continuation \( k \) implements the functional interface \( \text{Eff} \) with method \( \text{resume}() \text{ throws Effects} \). It closes over the initial stack segment \( \text{init} \) and, when invoked, prepends it to the stack \( s \). This implements the desired semantics of resuming the delimited continuation: The runtime system will first run the initial stack segment \( \text{init} \) before it eventually continues at the call site of \( \text{resume} \) within \( \text{body} \).

There are two casts involved that require some explanation. Both \( \text{shift} \) and the continuation \( k \) are effectful. Thus, the respective caller will be instrumented. However, by mutating field \( s \) and modifying the stack, we change the execution context. In \( \text{shift} \) we remove the initial segment of the stack and thus the new caller expects a value of type \( E \) not \( A \). In the continuation we prepend the initial segment and thus the caller now expects an \( A \) not \( E \). The Java type checker is unaware of our transformation and the modifications to our own call stack. Hence the casts\(^{24}\).

5.2.3 Step 3: Implementation of the JavaEffekt Library

In the previous subsections, we have seen how programs, which contain throws Effects annotations, are CPS translated. To support multi-prompt delimited continuations, we extended the runtime environment in which those translated programs are executed. Equipped with multi-prompt delimited continuations, we are now ready to implement the effect handler library.

The expressive power of effect handlers comes from the two operations handle and use, which are encapsulated in the library class Handler. Figure 5.7 shows the implementation of these two operations in terms of delimited continuations. As in the Scala implementation, handlers contain an instance of a prompt marker \( \text{Prompt}<E> \) as a field. The answer type of delimited continuations thus will be the effect domain \( E \). In the implementation of handle, we push the prompt before resuming with \( \text{prog} \). The pushed prompt will delimit the extent of continuations captured by that handler. By calling \( h.\text{unit} \) after resuming, programs that use no effects will be lifted from \( R \) to \( E \). The method use simply forwards to \( \text{shift} \). It again uses the prompt to capture the continuation up to the most recent call to handle for this handler instance.

\(^{24}\)Since \( A \) and \( E \) are generic type parameters, they will be erased and the program can safely be executed.
5. Java Effekt – Effectful Programming in Direct Style

abstract class Handler<R, E> {
  E unit(R r) throws Effects;

  private Prompt<E> prompt = new Prompt<E>() {};

  <A> A use(CPS<A, E> body) throws Effects {
    return Effekt.shift(prompt, body);
  }

  static <R, E, H extends Handler<R,E>> E handle(H h, Eff<H,R> prog) throws Effects {
    return Effekt.resetWith(h.prompt, () → h.unit(prog.resume(h)));
  }
}

Figure 5.7. The essence of the effect handler library: The Handler class.

5.3 Use Cases

Having implemented effect handlers as a library for Java, programmers can now freely combine object-oriented Java programming with effect handlers. We revisit selected examples of Section 4.4 to show how they can be expressed in JavaEffekt. Two differences to ScalaEffekt are particularly interesting. Firstly, in JavaEffekt, control-flow structures like conditionals and while loops can immediately be used in direct style. In ScalaEffekt, control flow needs to be lifted into the Control monad. Secondly, function-local (mutable) state (i.e., local variables) can be used and has the correct backtracking behavior in presence of multiple resumptions.

5.3.1 Handling Multiple Effects in one Handler

All handlers we have seen in this chapter so far only implemented a single effect signature. However, sometimes it is necessary to group the implementation of multiple effect signatures in a single handler. Since effect signatures are interfaces and handlers are classes implementing those interfaces, this is straightforward. Like with ScalaEffekt, effect implementations grouped in a single handler share the same effect domain E, share the private state of the handler and they can be implemented in terms of each other. In particular, sharing the effect domain is important if the handler wants to express interaction between different effect operations. Two examples combining Amb and Exc in one handler to share the effect domain are:

```java
class Nondet<R> extends Collect<R> implements Exc {
  <A> A raise(String msg) throws Effects {
    return use(k → Lists.empty());
  }
}
```
5.3. Use Cases

class Backtrack<R> extends Maybe<R> implements Amb {
  boolean flip() throws Effects {
    return use(k -> {
      Optional<R> res = k.resume(true);
      return res.isPresent() ? res : k.resume(false);
    });
  }
}

Handler Nondet extends AmbList and only provides the definition for raise. It shares the effect domain List<R> with AmbList. Similarly, handler Backtrack extends Maybe and adds the implementation for flip. It shares the effect domain Optional<R> with Maybe. By subtyping, the combined handlers can still be used as handlers for the individual effects Amb or Exc:

Handler.handle(new Nondet, nd -> drunkFlip(nd, nd))
▶ ["Heads", "Tails"]

Handler.handle(new Backtrack, bt -> drunkFlip(bt, bt))
▶ Optional["Heads"]

This also illustrates reuse of handler implementations by inheritance. We only needed to provide the missing definitions; all other implementations of effect operations are inherited.

5.3.2 Alternatives to Capability Passing

In previous examples, effectful methods were written in capability-passing style. They thus expressed their use of effects by expecting handler instances (capabilities) as arguments. In an object-oriented programming language like Java, it is natural to explore other means to get access to a handler instance. Capabilities can be passed to constructors and stored in fields. Take the following implementation of a reader effect (specialized to characters) as an example:

interface Input { char read() throws Effects; }

class StringInput<R> extends Handler<R, R> implements Input {
  final Exc exc; final String input; int pos = 0;
  StringInput(String s, Exc e) { this.input = s; this.exc = e; }
  public char read() throws Effects {
    return if (pos >= input.length()) exc.raise("EOS");
    else input.charAt(pos++);
  }
}

Methods of StringInput, which use the Exc effect, can only be safely executed in the corresponding dynamic scope of the Exc-handler. In JavaEffekt this is up to the user of StringInput to ensure. The next chapter revises our Scala library to statically guarantee effect safety.
5. Java Effekt – Effectful Programming in Direct Style

5.3.3 Ambient State and Parametrized Handlers

Our bytecode transformation only saves and restores function local state, that is, mutable local variables. It does not deep-copy heap-allocated state and nothing prevents the user from using mutable fields in the implementation of handlers. The handler `StringInput` is an example since it mutates the field `pos`. However, as demonstrated in Section 4.3, heap-allocated state and delimited continuations can interact in unforeseen ways. In JavaEffekt, we support ambient state as follows: Handler implementations, like `StringInput`, can just use mutable fields. To turn the handler state into ambient state, a handler needs to implement the `Stateful<S>` interface below.

```java
interface Stateful<S> { S exportState(); void importState(S state); }
```

```java
class StringInput2<R> extends StringInput<R> implements Stateful<Integer> {
    Integer exportState() { return pos; }
    void importState(Integer state) { pos = state; }
}
```

Our delimited control implementation is extended to export the handler state when the continuation is captured and to restore it on resumption. It is up to the user to identify and potentially deep-copy all relevant aspects of the handler state.

5.3.4 Case Study: Parsing

Equipped with nondeterministic choice, exceptions and reader we can implement parsers (Leijen, 2016). For convenience, we group `Amb`, `Exc`, and `Input` into one effect signature for parsers.

```java
interface Parser extends Amb, Exc, Input {}
```

```java
int digit(Parser p) throws Effects {
    char t = p.read();
    return isDigit(t) ? getNumericValue(t) :
                      p.raise("Not a digit: " + t);
}
```

```java
int number(Parser p) throws Effects {
    int res = digit(p);
    while (true)
        if (p.flip()) {
            res = res * 10 + digit(p);
        } else {
            return res;
        }
}
```

While we already presented a similar example in Section 4.6, implementing it again in JavaEffekt is interesting for two reasons. Firstly, it demonstrates that programs, now written in direct style, can also use Java’s control-flow constructs like `while` and conditionals. In the monadic counterpart of Effekt, those control-flow constructs need to be lifted into the monad. Secondly, it also illustrates that effectful programs can use mutable, function local state. To parse a number, every time after reading a digit, we mutate the local variable `res` to update the result. Our byte code transformation makes sure that this function local state will be saved on continuation capture, allowing a well-defined interaction with multiple resumptions and backtracking.

To parse the string "123" with the `number` parser, we use the `ParserForward` handler. The handler stores capabilities for `Amb`, `Exc` in `Input` in fields and implements the corresponding effect operations by forwarding. We can handle effect operations `flip` and `raise` with either the `Nondet` handler to get a list of all possible parses.

```java
Handler.handle(new Nondet<>(), nd ->
    Handler.handle(new StringInput2<=>("123", nd), r ->
        number(new ParserForward(nd, nd, r))));
▶ ["123", "12", "1"]
```
or with the Backtrack handler to only obtain the first successful parse, if it exists:

```
Handler.handle(new Backtrack<>(), bt →
    Handler.handle(new StringInput2<>("123", bt), r →
        number(new ParserForward(bt, bt, r)));
```

Note, that the parsers can now be written in direct style. At the same time, handlers (like Nondet) can still transparently access the continuation in their implementation of effect operations.

### 5.3.5 Case Study: Generators

In the programming language Python, the built-in control operation `yield` can be used to describe a stream of values also known as generators (Politz et al., 2013). While generators are built into Python, with effect handlers we can implement them as a library.

```java
interface Writer<A> { void write(A value) throws Effects; }
void numbers(int to, Writer<Integer> w) throws Effects {
    int n = 0; while (n ≤ to) { w.write(n++); }
}
```

The method `numbers` describes a generator that yields integers up to a given value using the Writer effect. Again, it uses a mutable variable `n` together with native control-flow structures (i.e., `while`). We can handle the writer effect with the Iterate handler. To suspend the generator until the next value is requested, the handler captures and stores the continuation on every write.

```java
class Iterate<A, R> extends Handler<R, IteratorEff<A>> implements Writer<A> {...}
interface IteratorEff<A> {
    boolean hasNext() throws Effects;
    A next() throws Effects;
}
IteratorEff<Integer> it = Handler.handle(new Iterate<>(), w → numbers(10, w));
while (it.hasNext()) { println(it.next()); }
```

Since the iterator is effectful, we cannot reuse the Java interface `Iterator`. Instead, the interface `IteratorEff` duplicates the interface and adds the `throws Effects` annotations. As a consequence, this prevents users from using Java’s `for` (`for (A a : iterator)`) syntax (Gosling et al., 2015), since it requires `iterator` to implement the “pure” iterator interface.

### 5.3.6 Case Study: Cooperative Multitasking

Like generators, cooperative multitasking and async/await can be implemented as a library (Dolan et al., 2017; Leijen, 2017a). Programs can use the Fiber effect (cf. Section 4.4) to fork and suspend processes. A process is an effectful program `Eff<Void, Void>`. Like in our implementation in Section 4.4, the Fiber effect makes use of Java’s default methods to implement the derived effect operation `forked`. The `Scheduler` handler in Figure 5.8 implements a round-robin scheduler that keeps a queue of all running processes in its mutable handler state. Suspending is implemented by enqueueing the continuation of the process and immediately returning. The handler will then pick the next process to execute.
interface Fiber {
    void suspend() throws Effects;
    boolean fork() throws Effects;
    void exit() throws Effects;
    default void forked(Eff<Void, Void> p) throws Effects {
        if (fork()) { p.resume(); exit(); }
    }
}

(a) The Fiber effect signature.

```java
Handler.handle(new Scheduler(), f -> {
    f.fork(() -> {
        println("world");
    });
    println("hello");
    f.suspend();
});
```

(b) User program using the Fiber effect.

class Scheduler extends Handler<Void, Void> implements Fiber {
    void suspend() throws Effects { use(k -> { queue.add(k); run(); }); }
    void exit() throws Effects { use(k -> null); }
    boolean fork() throws Effects {
        return use(k -> {
            queue.add() -> k.resume(true);
            queue.add() -> k.resume(false);
            run();
        });
    }
    private Queue<Eff<Void, Void>> queue = new LinkedList<>();
    private void run() throws Effects {
        while (!queue.isEmpty()) queue.remove().resume();
    }
}

(c) A round robin handler for the Fiber effect.

Figure 5.8. Effect signature Fiber with operations for cooperative multitasking and a round-robin scheduler implemented as handler Scheduler.
5.4 Implementation of the Type Selective CPS Transformation

Section 5.2.1 illustrated the type-selective CPS translation by example as a source-to-source transformation. In the current section, we now present the implementation of the translation on the level of bytecode and show how we handle control-flow constructs like jumps and exceptions. The translation is interesting in that it uses Java 8 closures to create continuations. This is different to other bytecode translations, which we will compare with in Section 5.5.1. We implemented the described transformation described using the OPAL framework by Eichberg and Hermann (2014) for static analysis and synthesis of JVM bytecode. Our implementation of the CPS transformation as well as all other components of JavaEffekt can be found online:

http://github.com/b-studios/java-effekt

We minimize the presentation to a relevant core set of language features and leave out details that distract from the essence of the transformation: Storing all necessary function-local state in closures and inserting calls to the API as described in Section 5.2. We model the JVM (Lindholm et al., 2015) and assume an abstract machine with registers (also referred to as locals, since they are function local), an operand stack (also referred to as operands, again function-local) and a frame stack (commonly referred to as stack). We use the term function-local state to refer to the values stored in locals and operands at a given time in the execution of a method. We adopt the JVM calling convention: function arguments are pushed on the operand stack by the caller, but accessible as locals by the callee, starting from register index 0. We assume every bytecode instruction is labeled, but omit labels that we never refer to. In our description of bytecode, labels are drawn from the set $\mathbb{L}$. We use sans-serif font to refer to object-language labels (e.g., init, or loop) and italic font for meta-language variables ranging over labels (e.g., label, or eff). Similarly, we use serif font to refer to method names, types etc. (e.g., doLoop, or MyExc) and italic font for meta-language variables ranging over names (e.g., name, or exc).

5.4.1 An Example with Jumps and Exceptions

We explain the transformation on an example method doLoop with the following Java source:

```java
boolean doLoop() throws Effects {
    Input r = Inputs.getInput(); // static method
    loop: try ( while ('\n' != r.read()) {} )
    catch (MyExc e) { return false; }
    exit: return true;
}
```

The bytecode of the method doLoop is shown in Figure 5.9a. The method doLoop performs effect calls to $r.\text{read}()$ until the result is either a newline or $r.\text{read}()$ raises a native exception. The example includes exception handling to illustrate, in detail, how the bytecode instrumentation interacts with native exceptions. As in the JVM, exception handling is modeled external to the list of bytecode instructions of a method and exception handlers are given in the form of regions:

```
excregion tryStartLabel tryEndLabel catchLabel exceptionType
```

In our example, let us assume an exception region excregion loop break catch MyExc is in place. That is, if an exception with a type name MyExc is raised in the dynamic region between the labels loop and break execution will be continued at label catch.
5. Java Effekt – Effectful Programming in Direct Style

\[
\text{method doLoop 1 throws Effects \{ }
\begin{align*}
\text{init : } & \quad \text{invoke Inputs.getInput 0} \\
& \quad \text{store 1} \\
\text{loop : } & \quad \text{const 'n'} \\
& \quad \text{load 1} \quad \text{// load Input from local 1} \\
\text{op : } & \quad \text{invoke Input.read 1} \\
& \quad \text{ifeq exit} \\
\text{break : } & \quad \text{goto loop} \\
\text{catch : } & \quad \text{const false} \\
& \quad \text{return} \\
\text{exit : } & \quad \text{const true} \\
& \quad \text{return} \\
\end{align*}
\]

\[
\text{excregion loop break catch MyExc}
\]

(a) Bytecode of method doLoop.

(b) Generated methods – method bodies in Fig. 5.9d.

\[
\text{effCalls(doLoop) = [init, op] \quad operands(init) = 0 \quad locals(init) = [0] \quad ep(init) = init} \\
\text{tmpLocal(doLoop) = 2 \quad operands(op) = 1 \quad locals(op) = [1] \quad ep(op) = ep$op}
\]

(c) Meta information as obtained by static analysis.

\[
\text{init : } \quad \text{invoke Inputs.getInput 0} \\
& \quad \text{store 1} \\
\text{loop : } & \quad \text{const 'n'} \\
& \quad \text{load 1} \\
\text{op : } & \quad \text{store 2} \quad \text{// save call operands} \\
& \quad \text{load 1} \quad \text{// load live locals} \\
& \quad \text{closure Frame.enter doLoop$op 2} \quad \text{// close over two values} \\
& \quad \text{invoke Effekt.push 1} \quad \text{// push closure to stack} \\
& \quad \text{load 2} \quad \text{// restore call operands} \\
& \quad \text{invoke Input.read 1} \\
& \quad \text{return$void} \\
\text{ep$op : } & \quad \text{load 0} \quad \text{// load arguments} \\
& \quad \text{load 1} \\
& \quad \text{store 1} \quad \text{// restore locals} \\
& \quad \text{invoke Effekt.result 0} \quad \text{// get results} \\
& \quad \text{ifeq exit} \\
\text{break : } & \quad \text{goto loop} \\
\text{catch : } & \quad \text{const false} \\
& \quad \text{return$void} \\
\text{exit : } & \quad \text{const true} \\
& \quad \text{return$void}
\]

(d) Result of translating the instructions of method doLoop.

\[
\begin{align*}
S[ \text{doLoop } ] &= \text{method doLoop 1 throws Effects \{ } \\
& \quad \text{load 0} \quad \text{// load 'this'} \\
& \quad \text{closure Frame.enter doLoop$init 1} \\
& \quad \text{const false} \quad \text{// load dummy value} \\
& \quad \text{return} \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{E}[ \text{doLoop$init } ] &= \text{method doLoop$init 1 \{ } \\
& \quad \text{goto init} \\
& \quad \text{...} \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{E}[ \text{doLoop$op } ] &= \text{method doLoop$op 2 \{ } \\
& \quad \text{goto ep$op} \\
& \quad \text{...} \\
\end{align*}
\]

\text{Figure 5.9. Example of translating a method doLoop}
5.4. Implementation of the Type Selective CPS Transformation

The syntax of the term-language is summarized in Figure 5.10a. For simplicity of the presentation, we do not concern ourselves with types and thus choose a uni-typed term-language. The translation does not distinguish instance methods and static methods. Hence, we only include a single method definition that consists of a name, a list of potentially raised exceptions and a list of labeled instructions. We omit the list of exception handler regions (the exception table) since it does not require any modification. The syntax of bytecode instructions $\text{Instr}$ in Figure 5.10a includes instructions to load constant values ($\text{const}$ $v$) to the operand stack, instructions to load from and store into function-local registers ($\text{load index}$, $\text{store index}$) and control-flow instructions ($\text{return}$, $\text{throws}$, $\text{ifeq}^{25}$ and $\text{goto}$). Finally, as with method declarations, we only include a single form of method invocation ($\text{invoke name arity}$) that subsumes static and virtual method calls. Virtual method calls take the receiver as the first argument and thus always have an arity greater than or equal to one. The syntax of instructions also includes an instruction $\text{closure}$, which we discuss in Section 5.4.3.

5.4.2 Translation of Methods

Only effectful methods that are marked as throwing Effects exceptions are translated. All other methods of a class are copied unchanged.

As can be seen in the source-to-source transformation in Figure 5.4, for one effectful method $m$, we generate multiple methods: a single method stub, using the translation function $S[m]$ and one entrypoint method for each effect call at label $\text{eff}$ using the translation function $E[m][\text{eff}]$ (both translation functions are defined in Figure 5.10b). Every effect call ($\text{eff} : \text{invoke name arity}$) inside a given method $m$ gives rise to an entrypoint uniquely identified by the label $\text{eff}$. The entrypoint itself is labeled $\text{ep(\text{eff})}$ and represents the continuation of the method $m$ after the effect call returned. For consistency, we also treat the initial entrypoint at label $\text{init}$ as effect call. The special entrypoint $\text{ep(\text{init})} = \text{init}$ refers to the label of the first original instruction of the method. By explicitly pushing the initial entrypoint, we perform trampolining for each effect call. For the example method $\text{doLoop}$, we thus generate three methods: the method stub $\text{doLoop}$ and the two entrypoint methods $\text{doLoop}_\text{init}$ and $\text{doLoop}_\text{op}$ (Figure 5.9b).

Rule T-Stub generates the method stub that first saves the local state and then immediately returns a dummy value. At that point, the local state only consists of the arguments supplied to the function call. As we will see shortly, $\text{saveState}$ thus pushes a closure that closes over the call arguments. When invoked, it resumes with the method corresponding to the initial entrypoint. For our example, this initial entrypoint method is $\text{doLoop}_\text{init}$. The returned result of the stub method will never be used; hence $\text{loadDummyResult}$ can load any constant value.

Rule T-Entrypoint takes a method $m$ and a label $\text{eff}$ to generate a new method corresponding to the entrypoint at $\text{eff}$. For our example, we generate two static entrypoint methods $\text{doLoop}_\text{init}$ and $\text{doLoop}_\text{op}$. The entrypoint methods use the same exception table as the original method $\text{doLoop}$. After the initial $\text{goto}$ instruction, the bodies of the generated methods are identical. For our example, the common part is given in Figure 5.9d. The only difference between the two generated methods is the initial jump. Since generating almost identical methods for each entrypoint leads to unnecessary growth of the class file, in our implementation of $\text{JavaEffekt}$, we perform dead code elimination after generating the bytecode. Using closures to save state, only saving live variables, and performing dead code elimination ultimately results in code, which (in spirit) is very close to handwritten code in continuation-passing style (as in Figure 5.4c).

25For our example, we assume $\text{ifeq label}$ pops two values and jumps to the given $\text{label}$ if the two values are equal.
5. Java Effekt – Effectful Programming in Direct Style

\[
\text{instr} \in \text{Instr} ::= \text{const} \, \text{Value} \mid \text{load} \, N \mid \text{store} \, N \mid \text{return} \mid \text{throw} \mid \text{goto} \, L \\
\text{method} \, \text{Name} \, \text{throws} \, \text{Name} \, \{ \text{Instr} \} \\
\text{label}, \, \text{eff}, \, \ldots \, \in \, \text{L} ::= \text{init} \mid \text{loop} \mid \text{op} \mid \ldots \\
\text{name}, \, \text{exc}, \, \ldots \, \in \, \text{Name} ::= \text{doLoop} \mid \text{Effekt} \text{.push} \mid \ldots
\]

(a) Syntax of methods and bytecode instructions.

\[
\text{(T-Stub)} \\
S[ \cdot ] : \text{Method} \rightarrow \text{Method} \\
S[ \text{method} \, \text{name} \, \text{arity} \, \text{throws} \, \text{Name} \{ \text{Instr} \} ] = \\
\text{method} \, \text{name} \, \text{arity} \, \text{throws} \, \text{Name} \{ \text{Instr} \} \\
\text{saveState}(\text{name}, \text{init}) \\\n\text{loadDummyResult} \\\n\text{return}
\]

(b) Transformation of effectful methods.

\[
\text{(T-Invoke-Eff)} \\
I[\cdot](\cdot) : (\text{L} : \text{Instr}) \rightarrow \text{Method} \rightarrow \text{L} : \text{Instr} \\
I[\text{eff}] \text{m} : \text{invoke} \, \text{name} \, \text{arity} \, \text{throws} \, \text{Name} \{ \text{Instr} \} \\
\text{saveCallOperands}(\text{tmpLocal}(\text{m}), \text{arity}) \\\n\text{saveState}(\text{methodName}(\text{m}), \text{eff}) \\\n\text{restoreCallOperands}(\text{tmpLocal}(\text{m}), \text{arity}) \\\n\text{invoke} \, \text{name} \, \text{arity} \, \text{throws} \, \text{Name} \{ \text{Instr} \} \\
\text{return} \text{void} \\
ep(\text{eff}) : \text{restoreState}(\text{eff}) \\
\text{invoke} \, \text{Effekt} \text{.result} \, 0
\]

(c) Transformation of bytecode instructions.

\[
\text{(T-Return)} \\
I[\text{label}] : \text{return} \, \text{m} = \\
\text{label} : \text{invoke} \, \text{Effekt} \text{.returnWith} \, 1 \\
\text{return} \text{void}
\]

\[
\text{(T-Other)} \\
I[\text{label}] : \text{instr} \, \text{m} = \text{label} : \text{instr}
\]

(d) Helper functions used to manage function local state.

Figure 5.10. Type selective CPS translation via bytecode transformation.
5.4.3 Saving Function Local State

To generate state saving and restoring code, we use the following information about a method \( m \), which we obtain by static analysis of the bytecode:

\[ \text{effCalls} : \text{Method} \rightarrow \mathbb{L} \]

the set of labels that mark effect calls, including the label \( e_{p_{\text{init}}} = \text{init} \) but no effect tail calls,

\[ \text{tmpLocal} : \text{Method} \rightarrow \mathbb{N} \]

the index of the first free register, not used by the original instructions of the given method.

Likewise, for each effect call \( \text{eff} \in \mathbb{L} \) in a method \( m \), the transformation uses the following information, which we again obtain by static analysis:

\[ \text{operands} : \mathbb{L} \rightarrow \mathbb{N} \]

the number of operands on the operand stack after the effect call (not including the result),

\[ \text{locals} : \mathbb{L} \rightarrow \mathbb{N} \]

the list of indices of local registers, which are live after the effect call,

\[ \text{ep} : \mathbb{L} \rightarrow \mathbb{L} \]

a unique label representing the entrypoint to jump to on resumption,

\[ \text{ep} : \text{Name} \times \mathbb{L} \rightarrow \text{Name} \]

for each pair of original method name and label – a unique method name for the corresponding entrypoint method.

We also define \( \text{epArity}(\text{eff}) \) to equal \( \text{operands}(\text{eff}) + |\text{locals}(\text{eff})| \), referring to the total number of values that need to be stored in the closure. That is, all operands and the number of locals, which are live after the effect call. The static information for doLoop is given in Figure 5.9c.

To actually save the state, the meta-function \( \text{saveState} \) generates code that stores those parts of the function-local state, which are necessary to resume the execution of the function. This includes all operands (after the effect call) and the contents of all registers, which correspond to live local variables. In the code generated by \( \text{saveState}(\text{doLoop}, \text{op}) \),

\begin{verbatim}
load 1 // load live locals
closure Frame.enter doLoop_op 2 // close over two values
invoke Effekt.push 1 // push closure to stack
\end{verbatim}

we can see that saving the state is achieved in three steps:

1. all live local variables are loaded to the operand stack; the operands do not need to be loaded since they are already on the operand stack
2. a new instance of a Frame is created as a lambda; the lambda uses the given method \( \text{ep}(\text{name}, \text{eff}) \) to implement the interface Frame.enter and closes over \( \text{epArity}(\text{eff}) \)-many values on the operand stack;
3. finally, the newly created frame is pushed using Effekt.push.
5. Java Effekt – Effectful Programming in Direct Style

In JVM bytecode, closures are created by issuing a specific `invokedynamic` call to a `lambda metafactory`. We refer to this call only in its specialized form as

```
closure interface name arity
```

The call to `closure` is provided with name of a functional interface (Gosling et al., 2015) `interface ∈ Name` (i.e., an interface with a single abstract method), a method `name ∈ Name`, which serves as the implementation of the interface, and an `arity ∈ N` that specifies the number of values the lambda should close over. The JVM runtime passes the closed-over values as additional arguments to the implementing method when the closure is applied.

5.4.4 Translation of Instructions

To generate the bodies of entrypoint methods, Figure 5.10c defines the translation function $I_m[: label : instr ]$. It specifies how a single labeled instruction is translated within a method $m$. The result of translating the body of `doLoop` can be found in Figure 5.9d.

Figure 5.10d gives the implementation of some of the helper functions that expand at translation time to generated code. Given a list of register indices, the unlisted functions `loadLocals` and `storeLocals` generate bytecode that loads from (respectively stores to) all given locals. We use the notation $x..y$ to denote a range of indexes from $x$ to $y$, both ends inclusive.

The translation of bytecode instructions behaves as identity (Rule $T$-Other) except for effect calls (Rule $T$-Invoke-Eff) and returns (Rule $T$-Return). To stress, non-effectful calls require no modification. To translate effect calls, rule $T$-Invoke-Eff saves the function-local state, performs the effect call, and suspends the method by returning to the trampoline (return void). In the translated program, all jumps to the effect call within $m$ should point to the instrumented call instead. Therefore, we change label `eff` to point to the first instruction of the state saving code. This also affects exception regions that mention `eff`, which do not require further modification. As in our `doLoop` example, the effect call might require arguments that reside on the operand stack at the time of state saving. To account for this, we use temporary locals (which will not be stored in the closure) to set the call operands temporarily aside.

The remainder of the function after the effect call is labeled with `ep(eff)`. Since it immediately follows a return, this part of the code is only reachable by the `goto ep(eff)` in the corresponding entrypoint method. At that time, all function state necessary for resumption has been passed as arguments and is thus stored in first `epArity(eff)`-many registers. For our example of `doLoop`, the code at label `ep(op) = epSop` assumes that one operand (the constant `\n`) and one local (an instance of type `Input`) have been passed as arguments and are thus available via registers 0 and 1. Before the function can be resumed, its state needs to be restored to where it has been left off. The meta-function `restoreState` loads all saved operands and locals (in this order) to the operand stack. It then writes the locals to the original registers (in reverse order). The result of the previous effect call is obtained by `Effekt.result`.

If the previous effect call exited abnormally by throwing an exception, `Effekt.result` as implemented in Figure 5.5, will re-raise this exception. Since we already restored all operands and locals, the exception will be raised in the correct context and trigger the correct exception handlers. Being defined in terms of labels, our translation does not need to modify the exception table. The rule $T$-Return replaces every return with a call to `Effekt.returnWith` to install the second half of the special calling convention.

---

26https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html
27For the translation, `interface` will always be Frame.enter.
5.5 Discussion and Related Work

We discuss design decisions and related work. Existing implementations of libraries and languages for (algebraic) effect handlers are either translations to a high-level language or involve a custom runtime implementation. High-level implementations translate effect handlers into delimited continuations (Kammar et al., 2013; Kiselyov and Sivaramakrishnan, 2016), free monads (Kiselyov and Ishii, 2015) or perform a CPS translation into another high-level language (Leijen, 2017c; Hillerström et al., 2017). Other implementations require a custom runtime that supports stack manipulation (Bauer and Pretnar, 2015; Dolan et al., 2017) or setjump / longjump (Leijen, 2017b). In this chapter, we explored a new implementation technique for effect handlers in terms of a CPS transformation of bytecode. Similarly, the discussion focuses on other bytecode transformation.

5.5.1 Continuations on the Java Virtual Machine

We review related work on (delimited) continuations and CPS transformations in the context of the Java virtual machine. Experimental implementations that modify the JVM exist (Dragos et al., 2007; Stadler et al., 2009) or are under development (Pressler, 2017). While those specialized runtimes could potentially be used as backend for our effect handler library, here we will focus on library solutions. We compare our CPS transformation with three other Java projects that perform bytecode instrumentation. A library for fibers “Quasar” (Parallel Universe Software Co., 2013), a library for one-shot continuations “JavaFlow”\(^\text{28}\) (Silaev, 2015), and a library for coroutines “Coroutines” (Faghiihi, 2015).

Continuation instantiation Approaches to capture the continuation can be characterized by the point-in-time the continuation is materialized. CPS transformations create the continuation before the execution of an effectful call. This is how we implemented \texttt{JavaEffekt}. It is also the case for implementations of effect handlers that rely on CPS (Hillerström et al., 2017) and corresponding monadic implementations in eager languages like \texttt{ScalaEffekt}. Quasar also explicitly stores all function state, before entering an effect call. Another approach is to instantiate the continuation only when it is needed. Effectful functions can also use a special exception at runtime to signal that the continuation needs to be captured (Sekiguchi et al., 2001; Pettyjohn et al., 2005; Loitsch, 2007). Other alternatives to signal continuation capture are sum types (Kiselyov and Sivaramakrishnan, 2016) or global flags (JavaFlow, Coroutines).

Stack restoration All bytecode continuation libraries in Java that we are aware of resume a continuation by replaying all function calls. This way, the JVM stack is restored before the execution of the program continues. Similarly, Koppel et al. (2018) showed recently how delimited continuations can be expressed with exceptions and state by replaying all effectful function calls on resumption. This implementation technique simplifies integration with exceptions, stack traces, and debuggers. It also is a technical consequence of not having a first class representation of continuation frames. However, restoring the stack is linear in the depth of the stack since all function calls need to be replayed.\(^\text{29}\) In contrast, in \texttt{JavaEffekt} we explicitly reify each continuation frame as a closure. Upon resumption, we just enter the first frame without restoring the full

\(^{28}\)We compare \texttt{JavaEffekt} with a recently maintained fork of the equally named Apache Commons project. The original project is located at: https://commons.apache.org/sandbox/commons-javaflow

\(^{29}\)For Quasar, this observation has been made by Aleksandar Prokopec (Oracle Labs) – personal communication.
Java stack. While this helps to reduce the asymptotic complexity from quadratic to linear, stack traces in JavaEffekt only show very few frames, which can impede debugging.

Function state representation The state, necessary to later resume a suspended function, consists of the function-local state and an entrypoint label. It can be represented and stored in different ways. The entrypoint label can be encoded as a number that will be dispatched upon with a switch statement at the beginning of the method. This is commonly combined with storing the function local data in a stack like data structure (Quasar, JavaFlow, Sekiguchi et al. (2001), Bierman et al. (2012)). An alternative is to replace the switch by dynamic dispatch and to store the function data in a closure (Pettyjohn et al., 2005). This is how JavaEffekt is implemented.

Multiple resumptions We designed JavaEffekt in a way that continuations can naturally be resumed multiple times. In implementations supporting only one-shot continuations, state update can be destructive, which makes it easier to implement continuations efficiently (Dolan et al., 2015). While JavaEffekt maintains one global immutable runtime stack (as in Figure 5.5), Quasar and JavaFlow maintain one mutable stack per delimited continuation / fiber. In such a setting, multiple resumptions are implemented by deeply cloning the corresponding stack and all nested stacks before resuming. In JavaEffekt, function-local state is copied into immutable frames. In addition, stack segments are immutable and can be shared across multiple resumptions.

Marking effectful functions Effekt uses Java's checked exception mechanism as a (coarse grained) effect type system. Thus, all effect operations and consequently all effectful functions that transitively use effect operations are required to be marked as throwing a special Effects exception. The exception is only used as a marker to distinguish pure from effectful methods. It will never be thrown at runtime. JavaFlow requires the user to annotate methods with a @continuable annotation. Java Coroutines only treats those methods as effectful that take a value of type Continuation as an argument. Quasar offers the user the choice between checked exceptions and Java annotations.

5.6 Performance of Effekt

We report on some preliminary performance results. The evaluation of performance is split into two parts: A part on the CPS transformation and a part on Effekt as a library for programming with effect handlers. All benchmarks were executed on a 2.5 GHz Intel Core i7 with 16GB of memory using ScalaMeter (Prokopec, 2012), a state of the art JVM benchmarking library. Each benchmark consisted of 50 runs, executed in a warmed up JVM instance (JDK version 1.8.0_181).

The implementation of JavaEffekt slightly deviates from the formal presentation in Section 5.4. To avoid push-pop-enter cycles, the initial entrypoint is not explicitly pushed but inlined. We also show the performance results for a variant of JavaEffekt that implements several optimizations: Similar to the approach by Pettyjohn et al. (2005), our alternative implementation does not materialize continuation frames upfront, but only when needed. In addition, only methods that contain at least one non-tail effect call are instrumented. Like in our formal presentation, the alternative implementation also materializes continuation frames by creating closures. We refer to this variant as JavaEffekt_opt. The user programs using JavaEffekt do not need to be changed.
5.6. Performance of Effekt

![Graphs showing performance of Effekt](image)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>javaEffekt</th>
<th>JavaEffektopt</th>
<th>Coroutines</th>
<th>Quasar</th>
<th>JavaFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop (1M)</td>
<td>1.6 ±0.1</td>
<td>28.5 ±0.7</td>
<td>1.6 ±0.0</td>
<td>5.4 ±0.3</td>
<td>62.0 ±1.2</td>
<td>11.2 ±0.3</td>
</tr>
<tr>
<td>Loop* (1M)</td>
<td>1.6 ±0.1</td>
<td>118.2 ±3.1</td>
<td>47.1 ±1.5</td>
<td>45.7 ±1.6</td>
<td>200.2 ±2.8</td>
<td>172.5 ±2.2</td>
</tr>
<tr>
<td>Recursive (1K)</td>
<td>0.0 ±0.0</td>
<td>0.2 ±0.0</td>
<td>0.1 ±0.0</td>
<td>0.1 ±0.1</td>
<td>0.7 ±0.5</td>
<td>0.1 ±0.0</td>
</tr>
<tr>
<td>Recursive* (1K)</td>
<td>0.0 ±0.0</td>
<td>0.3 ±0.0</td>
<td>0.2 ±0.0</td>
<td>9.7 ±0.9</td>
<td>64.0 ±0.6</td>
<td>24.9 ±0.9</td>
</tr>
<tr>
<td>Skynet (2)</td>
<td>0.5 ±0.0</td>
<td>33.5 ±10.5</td>
<td>10.0 ±3.4</td>
<td>5.8 ±2.1</td>
<td>45.4 ±1.0</td>
<td>43.3 ±9.3</td>
</tr>
<tr>
<td>Skynet* (2)</td>
<td>0.5 ±0.1</td>
<td>48.3 ±1.1</td>
<td>43.4 ±13.2</td>
<td>6.4 ±2.4</td>
<td>70.8 ±1.3</td>
<td>30.4 ±0.7</td>
</tr>
</tbody>
</table>

* Variants that perform regular suspend and resume.

**Figure 5.11.** Performance of bytecode instrumentation libraries. Runtime in ms, lower is better.
5. Java Effekt – Effectful Programming in Direct Style

5.6.1 Performance of the Bytecode Instrumentation

We evaluate the performance of our CPS transformation comparing with Quasar in version 0.7.9 (Parallel Universe Software Co., 2013), JavaFlow in version 2.6.0 (Silaev, 2015), and Coroutines in version 1.4.2 (Faghihi, 2015). We also measure the overhead compared to a baseline that does not capture continuations. All libraries perform some sort of bytecode transformation to support capturing the continuation. Since each of the libraries targets a particular domain (coroutines / fibers), capturing the continuation also involves additional overhead specific to the target application. Where possible, we reduced this overhead by disabling features – focusing on the continuation capturing aspect, only. As an example, the measurements for the Quasar library where executed using a trivial single threaded scheduler. The results of the measurements can be found in Figure 5.11.

To assess the instrumentation overhead, the Loop benchmark counts down from a given number to zero, performing some computation work at each step but not capturing the continuation. We can see that most of the overhead of creating continuation frames is eliminated in the alternative JavaEffekt opt. Java Coroutines save the function local state in arrays before entering a potentially suspending function call. This is unnecessary for the Loop benchmark, which does not suspend. The Loop* variant captures and immediately resumes the continuation at each step of counting down. Here the cost of storing the function state pays off for Java Coroutines, which now aligns with JavaEffekt opt. The baseline for Loop* does not suspend and is thus the same as for Loop.

To measure performance of capturing the continuation, the Recursive and Recursive* benchmarks also count down, but as recursive functions. For Recursive, we suspend the computation once before returning the result (at stack-depth n); correspondingly, for Recursive* we suspend once at every recursive call. Resuming continuations is linear in stack depth for all implementations but the two JavaEffekt implementations. In consequence, for the other implementations, Recursive* has a running time that is quadratic in n while it is linear for JavaEffekt and JavaEffekt opt. Resuming in our implementations is linear in the number intermediate prompts, while the other implementations are linear in the number of individual frames.

To measure the overhead of delimiting continuations, the Skynet benchmark30 recursively spawns $n$ fibers until $n = 2^{16}$ fibers are created at the leafs. In total, $\frac{n\log_2(n) + n - 1}{n - 1}$ fibers are created. We also refer to n as the fanout. Each fiber recursively spawns fibers, performs some computation, and finally aggregates the results. One fiber corresponds to one delimited continuation. The Skynet variant never suspends, but just creates the fibers which immediately return. The Skynet* variant in contrast suspends each fiber once before returning, resulting in a continuation capture and resumption per fiber. Quasar and JavaFlow maintain one stack per delimited continuation / fiber, each pre-allocating memory to store the function state. JavaFlow additionally maintains one stack per primitive type and copies the stack on every resumption. This leads to hundreds of thousands of arrays copies.31 The Coroutines library is optimized for one-shot continuations and large parts of the library are inlined in the generated bytecode. It also does not suffer from its linear stack restoration in the Skynet benchmark since the stack size of each fiber on suspension is at most one.

30https://github.com/atemerev/skynet
31In reaction to the first publication of our results (Brachthäuser et al., 2018), the authors of JavaFlow significantly improved the performance of their library in general, and one-shot continuations in particular. We report the updated results here.
To evaluate the performance of the overall framework, we compare our Java implementation of JavaEffekt with our implementation in Scala (ScalaEffekt, Chapter 4), and the effect library “Eff” (Torreborre, 2016), which we refer to as “Scala Eff” to avoid confusion with the language Eff by Bauer and Pretnar (2015). Scala Eff is a library for functional programming with effects, based on extensible effects and freer monads (Kiselyov and Ishii, 2015). The results of the measurements can be found in Figure 5.12.

The Stateloop Benchmark uses a state effect to count down from a given number \( n \). To allow a better comparison with Scala Eff, both of our library implementations use the functional state translation of Kammar et al. (2013) instead of our Stateful interface. The baseline implementation directly uses mutable state.

The CountdownFlip benchmark layers one ambiguity effect over eight state effects (Kiselyov and Ishii, 2015). It uses the state to count down from 1,000 and flips once before returning. The baseline does not model ambiguity but always returns true for \( \text{flip} \).

NQueens is an effect library benchmark from the literature (Kammar et al., 2013). For the baseline, we adopted a Java implementation found online\(^{32}\). The baseline makes use of mutable state and arrays, whereas the other four implementations use control effects and immutable lists.

In both state benchmarks, Stateloop and CountdownFlip, our implementations use the functional state translation. Of course we could also use our specialized state interface of Section 5.3.3 instead. Using the Stateful interface in the CountdownFlip benchmark, for JavaEffekt and eight layers, the measured run time goes down to 0.22ms ± 0.02ms. Similarly, for JavaEffekt\(_{opt}\), we measured 0.08ms ± 0.01ms – a speed up of about 46x compared to using functional state.

\(^{32}\)https://rosettacode.org/wiki/N-queens_problem
5. Java Effekt – Effectful Programming in Direct Style

The benchmarks indicate that the Java implementations of JavaEffekt are on par with ScalaEffekt. Compared to Scala Eff, our implementations offer speed ups of 1.4-16x using functional state and of 470x using our specialized state interface. Besides specializing state, we account the biggest performance over Scala Eff, that tail resumptive operations can be dynamic method calls in Effekt. A similar finding has been reported by Leijen (2017b). Compared to ScalaEffekt, performance improvements might be related to inlining the monadic Scala code by bytecode instrumentation and only capturing the continuation on demand (JavaEffekt\textsubscript{opt}).

5.7 Chapter Conclusion

We presented the first library for programming with effect handlers in direct style in Java. We showed how such a library can be implemented in terms of a continuation-passing style transformation and multi-prompt delimited continuations. Our continuation-passing style transformation allows trampolining, multiple resumptions and is competitive in its performance.
Chapter 6

Scala Effekt – Effect Safety through Regions

The implementations of Effekt, as presented in the previous chapters, do not guarantee effect safety. Capabilities can leak, which in turn leads to runtime errors.

In this chapter, we improve the involved (effect) types and present a variant of ScalaEffekt that supports effect safety and effect polymorphism. Our effect system guarantees that all effects are eventually handled and runtime errors caused by leaked capabilities are statically ruled out.

To the best of our knowledge, Effekt, as presented in this chapter, is the first library implementation of effect handlers that supports effect safety and effect polymorphism, without resorting to type-level programming. We describe a novel way of achieving effect safety in a library embedding by using intersection types and path-dependent types. The effect system of our library design fits well into the programming paradigm of capability passing and is inspired by the effect system of Zhang and Myers (2019). Capabilities carry an abstract type member, which represents an individual effect type and reflects the use of the capability on the type level. Handlers introduce capabilities and remove components of the intersection type. Reusing the existing type system of Scala, we get effect subtyping and effect polymorphism for free.

Let us recall our running example using our ScalaEffekt library, presented in Chapter 4:

```scala
def drunkFlip(amb: Amb, exc: Exc) = for {
  caught ← amb.flip()
  heads ← if (caught) amb.flip() else exc.raise("Too drunk")
} yield if (heads) "Heads" else "Tails"
```

As before, effectful methods mention the effects they use as additional arguments (i.e., capabilities). Handlers introduce a dynamic scope in which the corresponding capability can be used – the handler region. However, it is not guaranteed that methods only acquire capabilities by means of capability passing. For example, capabilities can be stored in references to be used...


6. Scala Effekt – Effect Safety through Regions

later – even outside of the handler region. This can lead to runtime exceptions and is a source of unsafety.

In this chapter, we solve the problem of effect safety and present a variant of ScalaEffekt that statically prevents programs from using leaked capabilities outside of the handler region. We repeat user programs of Chapter 4 and change their types, which are more precise now. This also becomes visible in the inferred type of `drunkFlip`:

```scala
def drunkFlip(amb: Amb, exc: Exc): Control[String, amb.eff & exc.eff]
```

We describe a novel way of achieving effect safety in a library embedding by using intersection types and path-dependent types (Odersky et al., 2006). The effect system of our library design fits well into the programming paradigm of capability passing and is inspired by the effect system of the \(\lambda_{\text{seq}}\)-calculus by Zhang and Myers (2019). As we will see in Section 6.1, we represent effect rows as the contravariant intersection of effect types, where an individual effect of a capability (like `amb`) is represented by the abstract type member `amb.eff`\(^{33}\). Handlers remove components of the intersection type. By reusing the existing type system of Scala, we get effect subtyping and effect polymorphism for free.

By the nature of an embedding into a practical, but unsound host language (Amin and Tate, 2016), type- and effect-safety of our library can only be guaranteed up to soundness of the host language. Furthermore, we do not present formal proofs of soundness and safety, but the design for a library implementation as an embedding into an existing, mainstream programming language. Our effect system is based on \(\lambda_{\text{seq}}\) – but we leave a formal treatment and corresponding soundness proofs to future work. To enable a more direct comparison, Section 6.6.1 presents a library embedding of the \(\lambda_{\text{seq}}\)-calculus (Zhang and Myers, 2019) into Scala. While we do not present formal proofs, from our experience of working with the library embedding, we are confident it inherits the following properties from \(\lambda_{\text{seq}}\):

**Effect safety** We embed an effect system, which makes sure that attempting to run a program with unhandled effects lead to Scala type error. For example, we can only call `run` on a program when all effects are handled and we reject programs like `run { amb.flip() }`. Like with systems based on monadic regions (Launchbury and Sabry, 1997; Moggi and Sabry, 2001; Kiselyov and Shan, 2008), capabilities can `leave` their defining handler scope. Our effect system only ensures that they cannot be `used` outside their defining handler scope.

**Effect subtyping** We use Scala’s support for subtyping of intersection types to implement effect subtyping. A program with type `Control[Int, exc.eff]` can be used where a program of type `Control[Int, exc.eff & amb.eff]` is expected.

**Effect polymorphism** We use Scala’s support for polymorphism to express effect polymorphic functions like, such as the following function adapted from Rytz et al. (2012):

```scala
def mapM[A, B, E](ls: List[A], f: A ⇒ Control[B, E]): Control[List[B], E]
```

Here, `mapM` is polymorphic in the effects `E` used by function `f`.

\(^{33}\)Despite being written lowercase, `effect` is not different from other type members. We just render it as a keyword to highlight its importance for our approach.
Existing implementations of languages with effect handlers either completely lack a static effect system – this includes Multicore OCaml (Dolan et al., 2014), Eff (Bauer and Pretnar, 2015), embeddings of Eff in OCaml (Kiselyov and Sivaramakrishnan, 2016), and previous versions of Effekt in Scala (Chapter 4) and Java (Chapter 5) – or they do not have sufficient support for effect polymorphism (Kammar et al., 2013; Inostroza and van der Storm, 2018). Languages and libraries with effect systems like Extensible Effects (Kiselyov et al., 2013), Koka (Leijen, 2017c), Links (Hillerström et al., 2017), Frank (Lindley et al., 2017), and Helium (Biernacki et al., 2019) require explicit lifting annotations to encapsulate effects in effectful higher order functions, like the function `mapM` above. Without such manual liftings, the implementation detail of effects used within `mapM` would leak into its type signature. In contrast, our effect safe version of ScalaEffekt requires no such manual lifting.

In summary, the contributions of this chapter are:

- For our implementation, we build on the operational semantics of Dybvig et al. (2007). We achieve effect safety by generalizing techniques of Launchbury and Sabry (1997) to nested regions (Kiselyov and Shan, 2008), and we use intersection types of abstract type members (Parreaux et al., 2017) to express nesting of regions. Our effect system rules out some use cases of multi-prompt delimited control (Kobori et al., 2016), but can express many interesting use cases of effect handlers.

- We implement ScalaEffekt as a thin layer on top of multi-prompt delimited continuations. Importantly, we demonstrate how effect safety for effects and handlers follows from the newly gained effect safety of our implementation of multi-prompt delimited continuations.

- We evaluate desirable properties like effect safety and effect parametricity of our library embedding. We discuss limitations and potential solutions.

- To evaluate the expressive power of our effect system, we revisit examples of Chapter 4. We show how they can safely be expressed. The programs do not have to change, only the (effect) types are more precise now.

- We discuss interesting opportunities to explore type- and effect-safe modularization of effectful programs, opened up by embedding Effekt into Scala.

The remainder of this chapter is structured as follows. Section 6.1 identifies multiple sources of unsafety. It then shows how to add effect safety to the implementation of delimited control as presented in Section 4.2. Section 6.2 presents an effect-safe variant of our interface for ambient state. In Section 6.3, we repeat the developments of previous chapters and express effect handlers as a thin library on top of (now effect-safe) delimited control. We revisit examples from previous chapters and show how to make them effect-safe. We discuss desirable properties of our effect-system and potential threats in Section 6.5. Section 6.4 discusses the combination of object-oriented programming and effect-safe handlers. Effect safety surfaces interesting new encapsulation challenges. Section 6.6 discusses related work and concludes.
6. Scala Effekt – Effect Safety through Regions

6.1 Effect-Safe Delimited Control

In Section 4.2, we have seen a version of Control that has one type parameter Result. By also indexing the type Prompt with a type parameter Result, we statically track answer types and guarantee that capturing and calling the continuation is type-safe (Gunter et al., 1995). However, this version of Control is not effect-safe: capabilities can leave their handler region. In particular, capabilities can close over prompts. Using a prompt outside of the dynamic scope of the corresponding reset leads to a runtime error. We identify two ways to leave the scope of reset.

1) Leaving the scope by returning  We can leave the scope of reset by returning from it Osvald et al. (2016). In this case it is possible to leak the prompt either through the heap

```scala
var p: Prompt[Unit] = null
val problem1 = run {
  for {
    _ ← reset { prompt ⇒ p = prompt; pure(()).}
    _ ← shift(p) { resume ⇒ pure(()). } // Exception: Prompt not found
  } yield ()
}
```

or by simply returning it as a result:

```scala
val problem2 = run {
  for {
    p ← reset { p ⇒ pure(p). }
    _ ← shift(p) { resume ⇒ pure(()). } // Exception: Prompt not found
  } yield ()
}
```

As observed by Osvald et al. (2016), both sources of leakage can also occur indirectly through functions or objects that close over the prompt. Prompts might even leave the scope of the enclosing run to then be used in the scope of a different run. Dybvig et al. (2007) use rank-2 types to prevent this particular source of error, but leave others to future work.

2) Leaving the scope by shifting  We can also leave the scope of reset by means of control effects.

```scala
val problem3 = run {
  reset { p ⇒
    shift(p) { resume ⇒
      shift(p) { resume ⇒ pure(()). } // Exception: Prompt not found
    }
  }
}
```

Due to our choice of -shift-, shifting to a prompt removes the enclosing reset. Consequently, shifting a second time inside the body of shift results in a runtime error. The evaluation context of the second call to shift is run { □} and does not contain any delimiters. Danvy
and Filinski (1990) operationally prevent this kind of runtime error by leaving the outer `reset` behind. However, in the setting of multiple prompts this is not sufficient. The delimiter can also be removed by shifting to a different prompt:

```scala
val problem4 = run {
  reset { p1 ⇒ reset { p2 ⇒
    shift(p1) { resume ⇒
      shift(p2) { resume ⇒ pure({}) } } // Exception: Prompt not found
    }
  }
}
```

We now introduce our implementation of an effect system that rules out the above four problematic programs, prevents the use of escaped capabilities, and guarantees effect safety. The underlying problem our effect system solves is a general one: we need to restrict the lifetime of a resource (capabilities in our case) to a certain dynamic region (the call to `reset` in our case). This problem also occurs in the domain of region based resource management (Kiselyov and Shan, 2008), object capabilities (Haller and Loiko, 2016), delimited control (Dybvig et al., 2007), scope safety in type-safe meta programming (Parreaux et al., 2017), as well as with prompt based implementations of effect handlers (as in Chapters 4 and 5). Our effect system is inspired by the $\lambda_{\varnothing}$ calculus (Zhang and Myers, 2019), which uses dependent types to track the set of used labels (i.e., prompts and capabilities) in the effect type. In Section 6.5, we discuss some properties of our effect system embedding, but leave formal proofs of safety and soundness to future work. For a better comparison, Section 6.6.1 gives a more immediate embedding of $\lambda_{\varnothing}$ into Scala.

### 6.1.1 Tracking and Delimiting Prompt Usage

Following Zhang and Myers (2019), our effect system builds on the idea of tracking the set of prompts that a program uses in its type. We represent prompts on the type level by their *singleton type* (`p.type`, for some prompt `p`).

In Scala, a singleton type is only inhabited by that very value it refers to. Two (path-dependent) singleton types (`p1.type` and `p2.type`) are equal, if and only if, their prefix paths (`p1` and `p2`) are stable and can be unified (Odersky et al., 2003; Odersky and Zenger, 2005b). That is, the type checker treats `p1.type` and `p2.type` as equal types, if and only if, it can show that `p1` and `p2` refer to the same immutable binding. This way, we prevent leakage via mutable references as in `problem1`.

We use Scala’s intersection types to describe a set of prompts. For example, we would model the term level set of prompts

```scala
val ps = Set(p1, p2, p3) by the type  type Ps = p1.type & p2.type & p3.type
```

Term-level set union is represented as type-level intersection. We call programs that use prompts *effectful* and consequently refer to types like `Ps` as “effect types” or simply as “effects”. As we will see, effectful programs are contravariant in their effect type. The top of Scala’s subtyping lattice (`Any`) describes the empty set. To emphasize this, we define the type alias

```scala
type Pure = Any
```
6. Scala Effekt – Effect Safety through Regions

```scala
trait Control[+A, -Effects] {
    def map[B](f: A ⇒ B): Control[B, Effects]
    def flatMap[B, FX](f: A ⇒ Control[B, FX]): Control[B, FX & Effects]
    def andThen[B, FX](c: Control[B, FX]): Control[B, FX & Effects]
}

def pure[A](value: ⇒ A): Control[A, Pure]
def run[A](c: Control[A, Pure]): A
```

Figure 6.1. The effect-safe control interface – changes compared to Figure 4.2 are highlighted. Type alias Pure defined in text.

and pure programs thus have an effect type Pure. Again, it is important to point out that the purity only refers to delimited control. Programs with a pure effect row still can use side effects like writing to files or accessing the network. The intersection of singleton types might not be inhabited, but this is irrelevant for our use case. We only use the intersection type as a phantom type to track used effects.

The Control Interface

To enable tracking of prompts (and effects in extension), Figure 6.1 revises the type Control and adds a second type parameter Effects. As before, the type Control[+A, -Effects] is marked as covariant in its first type parameter A. It is also marked as contravariant (Odersky et al., 2006) in its second type parameter Effects to obtain the correct subtyping relation between effectful programs. This also aligns well with our view that capabilities are obligations to the caller. By contravariance, pure programs (that do not use any prompts) are a subtype of effectful programs (that do use a non-empty intersection of prompt types). As an example, assuming prompts p₁, p₂, and p₃, we write the type of the effectful program that uses prompts p₁ and p₂ to compute an integer as:

```scala
val prog₁: Control[Int, p₁.type & p₂.type]
```

By (effect) subtyping, the effect type can be weakened

```scala
p₁.type & p₂.type & p₃.type <: p₁.type & p₂.type
```

and thus the following assignment is valid:

```scala
val prog₂: Control[Int, p₁.type & p₂.type & p₃.type] = prog₁
```

The type of Control.map shows that mapping pure functions does not add any new effects to the set of effects. In contrast, the function passed to Control.flatMap can have additional effects FX to produce a value of type B. The effects are aggregated using the intersection type FX & Effects. Lifting computation into the control monad with pure has no control effects. Dually, only programs without any unhandled control effects can be executed using run.
6.1. Effect-Safe Delimited Control

trait Prompt[Result, Effects] {}

def reset[R, FX](prog: (p: Prompt[R, FX]) ⇒ Control[R, p.type & FX]): Control[R, FX]
def shift[A, R, FX](p: Prompt[R, FX]):
    body: (A ⇒ Control[R, FX]) ⇒ Control[R, FX]
): Control[A, p.type]

Figure 6.2. The effect-safe prompt interface – changes compared to Figure 4.4 are highlighted.

6.1.2 The Prompt Interface – From Answer Type Safety to Effect Safety

In the previous variant of Control (Chapter 4), prompts carried the answer type Result to ensure that using control effects is type-safe (Gunter et al., 1995). To also make them effect-safe and prevent programs like problem4 from type checking, the type Prompt (Figure 6.2) now additionally contains a type parameter Effects.

As before, prompts are introduced by reset.

def reset[R, FX](prog: (p: Prompt[R, FX]) ⇒ Control[R, p.type & FX]): Control[R, FX]

There are a few important aspects of our embedded effect system that become visible in this type signature:

Ambient effects

The type parameter FX describes the set of ambient effects, which are available at the call site to reset. Not surprisingly, the return type of the delimited program prog also signals that the program can use these effects. Prompts now track both, the expected return type Result and the set of ambient effects Effects available at the corresponding reset.

Prompt usage and intersection types

The type of prog is a dependent function type (Odersky, 2019b). As before in Section 4.2, the program receives the fresh prompt as argument, which it may use in its body. However, this fact is now also expressed in the program’s return type Control[R, p.type & FX].

Path dependency and safety

To guarantee safety, we have to make sure that the only way to remove a prompt type from the intersection type is by delimiting the program with reset. To achieve this, the set of effects of the program prog refers to p.type, a path-dependent singleton type (Odersky and Zenger, 2005b). Different calls to reset lead to different singleton types that cannot be unified on the type level (even if it would be the same object at runtime). Hence, only the very call to reset that introduced a prompt can remove the prompt’s singleton type from the effect type.

The return type of the control operator shift expresses the use of the prompt on the type level:

shift(p)(...): Control[A, p.type]

Tainting the set of effects with the singleton type rules out problematic programs program1 and program2. In case of the first program, the prompt is stored in a mutable reference and used
outside of the corresponding reset. In Scala, two (path-dependent) types are equal if and only if their prefix paths are stable and they can be unified (Odersky and Zenger, 2005b). Informally, a path is stable if it does not contain a mutable component. This way we prevent leakage via mutable references as in problem1. The second program is also statically ruled out, since run requires the set of effects to be empty. Furthermore, all attempts to delimit a program that uses the prompt will fail: The singleton type of the prompt will not unify with the new delimiter.

As mentioned in Section 2.3, the body of a shift is conceptually evaluated at the position of the corresponding reset. This is now reflected in its type, which is:

\[
\text{body}: (A \Rightarrow \text{Control}[R, FX]) \Rightarrow \text{Control}[R, FX]
\]

Both, the answer type \(R\) and the effects \(FX\) have to match with the ones at the corresponding reset\([R, FX]\). Thus, capability-passing style is not only essential for operationally delimiting control effects but also necessary to carry both the expected answer type as well as the available effects from the reset to the shift that uses the prompt. Since the body of shift has to return Control\([R, FX]\), it cannot shift to the same prompt (as in problem3). This would require a type of Control\([R, p\cdot \text{type} \& FX]\). The problematic program problem4 is ruled out too, since \(p_1\) has type Prompt\([\text{Int}, \text{Pure}]\). The body of the first shift needs to be pure and cannot use \(p_2\).

To emphasize, we do not prevent leakage of prompts. Instead, we taint the effect type whenever we perform shift on a prompt. This is reflected in the return type of shift (Figure 6.2). The type Control\([A, p\cdot \text{type}]\) indicates the use of the prompt \(p\) on the type level. While we do not prevent leakage, the type of run asserts that only pure programs can be executed. That is, all prompts have to be delimited and the intersection has to be empty (Pure). The way we achieve safety is similar to how rank-2 types can be used to enable type-safe monadic regions in Haskell (Launchbury and Sabry, 1997; Moggi and Sabry, 2001; Kiselyov and Shan, 2008). In those approaches, resources are indexed by a type parameter. To handle programs that use resources, the state handler runST then requires that the type index can be quantified over universally. Every user of a resource is “infected with the type of that state thread” (Launchbury and Sabry, 1997, p. 229). Similarly, using a prompt \(p\) taints the effect row with the prompt’s singleton type. By building on path-dependent types, we move the universal quantification from the type level to the term level.

6.1.3 Structured Programming with Effect-Safe Delimited Control

Equipped with our enhanced implementation of Control, we are now ready to revisit the examples from Section 4.2 and assign effect types.

Example 1 - Effect Typed

The first example (Danvy and Filinski, 1990) does not need to change. Only the type is a bit more precise:

```scala
val ex: Control[Int, Pure] = reset { p: Prompt[Int, Pure] ⇒
  shift(p) { k ⇒ k(100) flatMap { x ⇒ k(x) } } map { y ⇒ 10 + y }
} map { z ⇒ 1 + z }
```

It is now clear from the effect type that, after resetting, there are no more control effects left to delimit. We can thus safely run ex.
Example 2 - Effect Typed

The second example illustrates how each reset removes its prompt from the set of effects.

```scala
val ex2: Control[Int, Pure] = reset { p1: Prompt[Int, Pure] ⇒
  reset { p2: Prompt[Boolean, p1.type] ⇒
    shift(p1) { k ⇒ pure(21) } // Control[Int, p2.type & p1.type]
    } map { x ⇒ if (x) 1 else 2 } // Control[Int, p1.type]
  } map { y ⇒ 2 * y } // Control[Int, Pure]
```

The body of the second reset has type Control[Int, p1.type & p2.type]. By effect subtyping, we can use shift(p1) { ... }, which has type Control[Int, p1.type].

This example also highlights another important aspect of our effect-safe control operator: prompts track the available effects at the definition site, not the use site. For instance, the type of p2 informs us that within a body of shift(s2) { ... } the s1 prompt could be used. In this case, the return type of shift(s2) would only mention s2.type, not s1.type. The potential usage of s1 within the body is an implementation detail that is encapsulated at the definition site. It does not leak to the use site of shift(s2).

Example 3 - Effect Typed

Adding effect types to the third example is a bit more involved. In Section 4.2, we defined the following type aliases (Figure 4.6a):

```scala
type Amb = () ⇒ Control[Boolean]
type Exc = String ⇒ Control[Nothing]
```

Without effect safety, it was sufficient to say that flip and raise use any control effects by making them return for example Control[Boolean]. To establish effect safety, we now have to be more specific. Figure 6.3a defines the effect signatures Amb and Exc as traits. Comparing the traits to the equally named type aliases, we see a few differences:

1. Effect signatures inherit from the library trait Eff.
2. Effect operations are now named (that is, flip and raise are explicitly named methods) whereas earlier we used the apply method that Scala generates for function types.
3. Effect operations refer to the type member effect, declared in Eff, in their return type.
   Note that the reference to effect is short for this.eff (Odersky and Zenger, 2005b).

The last difference is especially important. Guided by the previous implementation, we know that a handler will eventually use some effect to implement the effect operations of the effect signature. The concrete effects used, however, are considered an implementation detail of the handler. We thus hide this detail existentially (Mitchell and Plotkin, 1988; Rossberg et al., 2010) behind a type member effect.

The use of these effects is now exactly as in the introductory example:

```scala
def drunkFlip(exc: Exc, amb: Amb): Control[String, amb.effect & exc.effect] =
  for {
    caught ← amb.flip()
    heads ← if (caught) amb.flip() else exc.raise("Too drunk")
  } yield if (heads) "Heads" else "Tails"
```
trait Eff { type effect }

trait Exc extends Eff { def raise(msg: String): Control[Nothing, effect] }

trait Amb extends Eff { def flip(): Control[Boolean, effect] }

(a) Effect Signatures for exception and ambiguity.

def maybe[R, FX](prog: (e: Exc) ⇒ Control[R, e.effect & FX]): Control[Option[R], FX] =
reset {( p: Prompt[Option[R], FX]) ⇒
    val exc = new Exc {
        type effect = p.type
        def raise(msg: String) = shift(p) { resume ⇒ pure(None) }
    }
    prog(exc) map { x ⇒ Some(x) }
}

(b) Handler function for the exception effect. Implementation of raise like in Figure 4.6b.

def collect[R, FX](prog: (a: Amb) ⇒ Control[R, a.effect & FX]): Control[List[R], FX] =
reset {( p: Prompt[List[R], FX]) ⇒
    val amb = new Amb {
        type effect = p.type
        def flip() = shift(p) { resume ⇒ for {
            xs ← resume(true)
            ys ← resume(false)
        } yield xs ++ ys }
    }
    prog(amb) map { x ⇒ List(x) }
}

(c) Handler function for the ambiguity effect. Implementation of flip like in Figure 4.6c.

Figure 6.3. Using effect-safe delimited control to declare and handle exception and ambiguity effects. Effect type related changes compared to Figure 4.6 highlighted.
The implementation of handler functions is given in Figures 6.3b and 6.3c. The implementations of `raise` and `flip` in the handler functions are exactly as in Figure 4.6, but we repeat them for easier reference. However, we now assign types that are more precise! Handlers have to state that they implement the effect signatures in terms of delimited control. This is achieved by the type assignment `type effect = p.type`, which is necessary to unify `amb.effect` with `p.type`. This way `reset`, which removes the `p.type` from the set of effects, also removes `amb.effect`.

To emphasize, the simple library trait `Eff` plays an essential role in abstracting over the (control) effects used by a handler. This implementation detail is hidden from the handled program. Its type only refers to the abstract type member `effect`, the concrete implementation in terms of delimited control is hidden existentially.

With effect types assigned to `maybe` and `collect`, we are ready to handle our running example:

```java
val res = run {
    collect { amb ⇒
        maybe { exc ⇒
            drunkFlip(amb, exc) // Control[String, exc.effect & amb.effect]
        } // Control[Option[String], amb.effect]
    } // Control[List[Option[String]], Pure]
} // List[Option[String]]
```

### 6.2 Effect-Safe Ambient State

As seen in Section 4.3, and pointed out by Kiselyov et al. (2006), the interaction between delimited control and mutable state is subtle. The functional state translation of Kammar et al. (2013) allows us to guarantee the correct backtracking behavior, even in presence of handlers that call the resumption multiple times. However, the benchmarks of Section 5.6.2 suggest that using our specialized state interface offers significant speedups over using functional state. Figure 6.4 defines the effect signature of our built-in state effect. Only the types are more precise, compared to Figure 4.7b. As before, the interface `Field` is nested in the state effect. However, it now refers to the outer effect’s type member `effect`. This way, all fields created by calling the method `Field` share the same effect type (Leijen (2018b) also refers to this parent effect as “umbrella effect”). Simply by nesting the types, we are able to express scoped resources, whose lifetime is conceptually coupled to the outer effect. This allows creating a dynamic number of fields, while maintaining the invariant that fields cannot escape the region of the corresponding state handler.

Given the built-in handler `region` (Figure 6.4), we can again assign more precise types to the equivalent example of Section 4.3:

```java
collect { amb ⇒
    region { state ⇒
        val x = state.Field(0)
        amb.flip() flatMap { b ⇒
            if (b) x.put(2) else pure(()
        } andThen x.get() // Control[Int, state.effect & amb.effect]
    } // Control[Int, amb.effect]
} // Control[List[Int], Pure]
```

139
6. Scala Effekt – Effect Safety through Regions

trait State extends Eff {
  def Field[T](init: T): Field[T]
}

trait Field[T] {
  def get(): Control[T, effect]
  def put(value: T): Control[Unit, effect]
  def update(f: T ⇒ T): Control[Unit, effect]
}

def region[R, FX](prog: (s: State) ⇒ Control[R, s.effect & FX]): Control[R, FX] = ...

Figure 6.4. Effect-safe state effect – effect signature of the built-in state effect and its handler region.

Like the delimiter for continuation capture (i.e., reset) creates a fresh prompt, the delimiter for ambient state (i.e., region) creates a fresh instance of State. This instance can only be used in the scope delimited by region, which is guaranteed by the types.

6.3 From Effect-Safe Delimited Control to Effect Handlers

In Section 6.1, we have seen how to make delimited control effect-safe. This allowed us to safely re-implement our running example in the style of structured programming with delimited continuations (Kammar et al., 2013). The handlers of the previous section were implemented as anonymous inner classes. This style of implementing handlers precludes certain forms of reuse.

What is left, is the small step from structured programming with delimited control to effect handlers. A step we have already taken twice in Sections 4.4 and 5.2.2 for our Scala and Java implementations, correspondingly. Analogous to the previous chapters, we can express the handler interface of Effekt in terms of effect-safe multi-prompt delimited control. As explored in Section 4.4, implementing handlers as traits opens up interesting new opportunities for modularity and extensibility.

In this section, we define the missing Handler interface, revisit a selection of examples of Chapter 4, and show how to implement them in our effect-safe variant of ScalaEffekt. Not only is it possible to implement all the examples of Chapter 4 in an effect-safe way, adding effect types also makes important details visible that were previously hidden.

6.3.1 Programming with Effect-Safe Effect Handlers

Many handler implementations need to capture the delimited continuation and use delimited control as an implementation effect. Figure 6.5b abstracts over this common mode of use in the effect-safe definition of the Handler trait.
6.3. From Effect-Safe Delimited Control to Effect Handlers

class Maybe[R, FX] extends Exc with Handler[R, Option[R]] { type Effects = FX; ... }
class Collect[R, FX] extends Amb with Handler[R, List[R]] { type Effects = FX; ... }

(a) The two effect handlers are parametric in the effects FX.

trait Handler[R, E] extends Eff {
  // Abstract members that need to be specified by implementing classes
  type Effects
  protected def unit(result: R): Control[E, Effects]
  protected def use[A]({
    body: (A ⇒ Control[E, Effects]) ⇒ Control[E, Effects]
  }): Control[A, effect] = ...
  def handle(
    prog: this.type ⇒ Control[R, effect & Effects]
  ): Control[E, Effects] = ...
}

(b) The effect-safe handler interface. The abstract type member Effects describes the effects the handler may use.

Figure 6.5. Implementation of effect handlers for Exc and Amb using the effect-safe library class Handler – changes compared to Figure 4.3 are highlighted.

The Effect-Safe Handler Interface

Most importantly, the library trait Handler declares the type member Effects, which represents the implementation effects a handler uses besides delimited control. As we will see, handler implementations override the type member to specify concrete effects. Inspecting the type signatures of the three methods (i.e., unit, use, and handle) sheds some light on how a handler can use the implementation effects.

The type of the method unit signals that the return clause can use the specified implementation effects, but not the handled effect itself (effect).

The type of the method use shows that the body, which is passed to it, can use the implementation effects. Since the implementation effects are not handled by this handler, the type of the continuation (A ⇒ Control[E, Effects]) also mentions Effects. As already pointed out in Chapter 2, effect encapsulation is one of the most important aspects of effect handlers. The effects at the handler site (Effects) are encapsulated by use whose return type only mentions the effect of the handler itself (effect). This effect encapsulation is the only difference between ambiently bound lambdas and ambient functions in Chapter 3.

Analogous to reset, the type of the method handle states that the current effect h.effect is removed from the set of the effects used by prog. The implementation effects are not affected by calling handle.

Figure 6.5 uses the effect-safe handler interface to, once more, implement the handlers Maybe and Collect. The implementation of the methods stays the same. Only the types change. The example handlers Maybe and Collect do not use any other effects. They are parametric in the ambient effects FX, much like they are parametric in the answer type R. While the result type is
6. Scala Effekt – Effect Safety through Regions

provided as a type parameter to the handler, for the effects we use the type member Effects. As we will see, this offers us some more flexibility, since type members can refer to other type members of the same class, while type parameters cannot.

The Effect-Safe Handler Implementation

As in the previous chapters, in our implementation of ScalaEffekt, we implement the effect handler interface by adding a field that stores a unique prompt marker:

```scala
val prompt: Prompt[E, Effects]
```

Unlike before, however, the type of the prompt marker now carries the additional information about the effects at the handler site (Effects). For convenient handling, we again extend our implementation of multi-prompt delimited continuations with the method `resetWith`. The method bodies of `use` and `handle` are thus syntactically equivalent to the ones of Figure 4.8. The `Handler` trait uses control-effects. In our effect-safe implementation, this also needs to be declared on the type level:

```scala
type effect = prompt.type
```

Using this type equality, the implementations of `use` and `handle` can be type checked without any modifications.

While with the `Handler` trait we aim to abstract over the implementation detail of multi-prompt delimited control, the type member `effect` and its definition are part of the public interface. Once fully implemented, the proposed Scala feature of opaque types will help us to fully encapsulate this implementation detail (Osheim and Cantero, 2017).

6.3.2 Case Study: Effect-Safe Cooperative Multitasking

Using our effect-safe handler interface, we can implement all the effects and handlers from the Chapter 4, but most implementations are straightforward and do not bare much new technical insight. However, two handler implementations surface interesting properties of our Effekt design and our effect system: the Poll handler (for the Fiber effect) and Scheduler handler (for the Async effect). To recall, the handler for the Fiber effect does not explicitly capture the continuation, but only uses other effects in its implementation. In contrast, the handler for the Async effect does capture the continuation and additionally uses another effect (i.e., State).

We now revisit these two effects (cf. Section 4.4) and handlers and add effect types. As we will see, adding effect types makes important details visible that were previously hidden.

The Scheduler Handler – Storing effectful computation

Figure 6.6 defines the effect-safe variants of the Fiber effect and its handler Scheduler. All term-level implementations are left unchanged but are repeated for convenience.

Adding effect types to the Fiber effect signature (Figure 6.6a) exhibits an interesting detail. The derived effect operation `forked` is a higher-order effect operation (Wu et al., 2014): it takes a computation as argument. While it is higher-order, the effect types inform us that it is not a handler. The computation passed as argument can have arbitrary effects FX. However, these effects are left unchanged. They still appear in the return type of `forked`. 

142
trait Fiber extends Eff {
  def suspend(): Control[Unit, effect]
  def fork(): Control[Boolean, effect]
  def exit(): Control[Nothing, effect]
  def forked[FX](p: Control[Unit, FX]): Control[Unit, effect & FX] = for {
    b ← fork()
    r ← if (b) p andThen exit() else pure(())
  } yield r
}

(a) The Fiber effect signature – the derived operation forked is polymorphic in ambient effects FX.

trait Scheduler[R, FX] extends Fiber with Handler[R, Unit] {
  val state: State
  type Effects = state.effect & FX
  def unit(r: R) = pure(())
  type Queue = List[Control[Unit, Effects]]
  lazy val queue = state.Field[Queue](Nil)
  def exit() = use { resume ⇒ pure(()). }
  def fork() = use { resume ⇒
    queue.update { resume(true) :: resume(false) :: _ } andThen run
  }
  def suspend() = use { resume ⇒
    queue.update { _ appended resume(()). } andThen run
  }
  private def run: Control[Unit, Effects] = queue.get() flatMap {
    case Nil ⇒ pure(())
    case p :: rest ⇒ queue.put(rest) andThen p andThen run
  }
}

(b) Handler for the Fiber effect – enqueued processes now have more precise types.

Figure 6.6. Effect-safe versions of the Fiber effect signature and the Scheduler handler. All term-level implementations are unchanged.
Also the types of the Poll handler (Figure 6.6b) are more precise now. By defining

\[
\text{type Effects = state.effect \& FX}
\]

the handler implementation declares that it uses the state capability. Specializing the type of use[A] to this handler implementation, we obtain for the argument body:

\[
\text{body: (A \Rightarrow Control[\text{Unit}, state.effect \& FX]) \Rightarrow Control[\text{Unit}, state.effect \& FX]}
\]

This shows two things. Firstly, the handler is allowed to use the state effect within the body passed to use. Secondly, continuations captured with use also still have the state effect and other ambient effects FX. One can easily make mistakes when implementing the scheduler example without effect types. As before, the scheduler stores continuations in a queue to later run them in a potentially different context (Dolan et al., 2017). The previous type of stored computations (i.e., Control[\text{Unit}]) did not provide much information about when it is safe to run. In the effect-safe implementation, the type of Queue needs to be more specific. It now contains computations of type Control[\text{Unit}, state.effect \& FX]. Importantly, since the computation is indexed by the involved effect types, it cannot be used outside of the handler for state. The handler function for the scheduler leaves the state effect open:

\[
\text{def scheduler[R, FX](st: State)(prog: (f: Fiber) \Rightarrow Control[R, f.effect \& FX]) =}
\]

\[
\text{new Scheduler[R, E] { val state: st.type = st } handle { fiber \Rightarrow prog(fiber) }}
\]

The type refinement val state: st.type is necessary for a precise return type, which is inferred to be Control[\text{Unit}, state.effect \& FX]. Without the type refinement, the state handler could not remove the state effect from the set of effects.

The Async Handler – Forwarding without Capturing

Also the improved types of the Async effect (Figure 6.7a) and its handler implementation (Figure 6.7b) reveal some interesting details. Like the derived effect operation forked, the effect operation async is higher-order and takes a computation. Its return type also mentions the effect parameter FX, which suggests that promises contain effectful computations. Those computations are conceptually evaluated at the original call site to async. Correspondingly, the signature of await mentions no effects beside this.effect.

As mentioned before, handlers in Effekt do not have to capture and use the continuation and consequently do not have to inherit from the Handler trait. It is up to the handler implementation to decide. Figure 6.7b repeats the implementation of the Poll effect handler but adds effect types. The only necessary change is the definition of the type member effect:

\[
\text{type effect = state.effect \& fiber.effect}
\]

Instead of extending the library trait Handler, the Poll handler directly implements the Async effect in terms of the two effects State and Fiber. This is not possible in languages like Koka, where every effect handler always has to capture the continuation. As before, the handler function poll takes the two required capabilities to construct an instance of the handler Poll:

\[
\text{def poll[R, FX](s: State, f: Fiber)(prog: (a: Async) \Rightarrow Control[R, a.effect \& FX]) =}
\]

\[
\text{prog(new Poll { val state: s.type = s; val fiber: f.type = f })}
\]
6.3. From Effect-Safe Delimited Control to Effect Handlers

```scala
trait Async extends Eff {
  type Promise[T]
  def async[T, FX](prog: Control[T, FX]): Control[Promise[T], effect & FX]
  def await[T](p: Promise[T]): Control[T, effect]
}
```

(a) The Async effect signature – promises contain effectful computations.

```scala
trait Poll extends Async {
  val state: State; val fiber: Fiber

  type effect = state.effect & fiber.effect
  type Promise[T] = state.Field[Option[T]]

  def async[T, FX](prog: Control[T, FX]) = for {
    p ← pure(state.Field[Option[T]])(None)
    _ ← fiber.forked { prog flatMap { r ⇒ p.put(Some(r)) } }
  } yield p

  def await[T](p: promise[T]) = p.get() flatMap {
    case Some(r) ⇒ pure(r)
    case None ⇒ fiber.suspend() andThen await(p)
  }
}
```

(b) Handler for the Async effect – using two effects State and Fiber.

**Figure 6.7.** Effect-safe versions of the Async effect signature and the Poll handler. All term-level implementations are unchanged.
6. Scala Effekt – Effect Safety through Regions

By refining the types of state and fiber to singleton types, the inferred return type of poll is Control[R, s.effect & f.effect & FX]. It thus communicates precisely that we implement the Async effect in terms of the given state and fiber capabilities.

We are now ready to safely handle the example program asyncExample from Section 4.4. The user program does not require any modifications.

```scala
region { s ⇒
  scheduler[Unit, s.effect](s) { f ⇒
    poll(s, f) { a ⇒
      asyncExample(f, a) // Control[Unit, f.effect & a.effect]
    } // Control[f.effect & s.effect]
  } // Control[s.effect]
} // Control[Pure]
```

The effect types illustrate that the poll handler function removes the async effect (a.effect), but adds the state effect (s.effect) to the set of effects. The scheduler handler function uses the same state effect but in turn removes the fiber effect (f.effect). Finally, the region handler handles the state effect. Type inference is not proficient enough to infer the removal of effects and we need to annotate the effect type s.effect at the call to scheduler.

This concludes our presentation of effect-safe programming with effect handlers. In this section, we have introduced the missing Handler interface, revisited some non-trivial example usages, and showed how they can be expressed in effect-safe ScalaEffekt.

6.4 Discussion: Effect Handlers and Object Orientation

Effekt is an embedding of effect handlers in a language with support for object-oriented programming. Naturally, the question arises how these two features interact. We will now revisit this combination in the light of our effect-safe implementation.

Object-oriented programming has a strong focus on encapsulation. In particular, the concrete implementation of an object and its internal state is often hidden behind an interface (Canning et al., 1989). That is, the implementation can differ with the granularity of a single object. Another important feature is that objects are first-class and are typically stored on the heap (Lindholm et al., 2015). In contrast, effects and handlers are tied to a stack discipline. Effect handler implementations can capture parts of the stack as a continuation, handlers delimit segments of the stack and effect typing asserts that these stack operations are safe.

Which effects are used by an object’s implementation can be seen either as part of the public interface or as a private implementation detail. It is a design decision the programmer should make. However, if the effects used by an object are hidden behind an interface, how can we assert effect safety? For instance, if an object closes over a capability, the object’s lifetime needs to be restricted to the capability’s lifetime (Osvald et al., 2016). Otherwise, the use of the capability within the object might not be effect-safe. In this section, we will discuss possible design choices when combining effect handlers with object-oriented programming, while maintaining effect safety. The following interface will serve as a running example:

```scala
trait Person {
  def greet(other: String): Unit
}
```
Alternative 1. Effects as Part of the Public Interface

An implementation of this interface might want to use the following effect to print the greeting on the console.

```
trait Console extends Eff { def print(msg: String): Control[Unit, effect] }
```

However, the method `greet` as defined above does not mention the `Console` effect. Of course, we can change the interface accordingly.

```
trait Person {
  def greet(other: String)(out: Console): Control[Unit, out.effect]
}
```

Now, the `Console` effect is part of the public interface and all implementations of `Person` can make use of it to implement method `greet`. The effect has to be handled by the caller of `greet`. In this variant, it is possible to have multiple implementations of `Person` and store the instances in data structures on the heap.

```
var p1: Person = new Person { ... }
var p2: Person = new Person { ... }
val ps = List(p1, p2)
```

Alternative 2. Hiding Effects behind an Interface

Changing the interface of `Person` to mention the effects, which are used by a particular implementation, leaks implementation details. This problem also occurs with checked exceptions in Java. We can think of `Console` as a checked exception that is not mentioned in the interface of `greet`. Java programmers often resort to wrapping checked exceptions in unchecked ones to work around this problem (Zhang et al., 2016). Sometimes, the exceptions our implementation throws are considered implementation details that we might want to encapsulate. Similarly, in Effekt, we can hide the effects behind an abstract type member `effect`.

```
trait Person {
  type effect
  def greet(other: String): Control[Unit, effect]
}
```

An implementation of `Person` closes over the effect capabilities, just like the handlers from the previous section.

```
class MyPerson extends Person {
  val out: Console
type effect = out.effect
  def greet(other: String) = out.print("Hello " + other)
}
```

This way, the lifetime of an object of type `MyPerson` is tightly coupled to the lifetime of the capability `out`. Let us assume some handler `withConsole` for the `Console` effect:
6. Scala Effekt – Effect Safety through Regions

```scala
withConsole { o ⇒
  ...
  val p = new MyPerson { val out: o.type = o }
  ...
}
```

The instance `p` must not be used outside the scope of the handler `withConsole`, which is ensured by our effect system: `out.effect` is an abstract type that only unifies with this one particular call to `withConsole`. As before, to eventually be able to handle the effects used by the implementation, users thus always need to have stable paths to an object. In the following example `p1` and `p2` are arguments of method `user`. They have stable paths that can be used in the return type.

```scala
def user(p1: Person, p2: Person): Control[Unit, p1.effect & p2.effect] = for {
  _ ← p1.greet("Alice")
  _ ← p2.greet("Bob")
} yield ()
```

In general, the requirement of path stability (Odersky and Zenger, 2005b) excludes objects to be stored in mutable references or in containers like lists. While we can store `p1` in a reference, the effect system will prevent us from running a program that calls an effectful methods on it.

### Alternative 3. Grouping Objects by their Effect Implementations

The first alternative requires all objects to use the same effects in their implementation and the second alternative allows each object to individually differ in their effect implementation. Both solutions also have drawbacks: the former constrains the implementer while the latter imposes restrictions on the user. As a compromise between the two extremes, we can generalize over the effect implementation and thereby group objects by their effect implementations.

```scala
trait Person[FX] {
  def greet(other: String): Control[Unit, FX]
}
```

Like with abstract type members, implementing classes can instantiate `FX` to the desired implementation effects. Like with the first alternative, objects of type `Person[out.effect]` leak the implementation detail that they use the `Console` effect in their implementation. Users can be parametric in the effect type of the particular implementation:

```scala
def user[FX](p1: Person[FX], p2: Person[FX]): Control[Unit, FX] = for {
  _ ← p1.greet("Alice")
  _ ← p2.greet("Bob")
} yield ()
```

While we now can store objects of type `Person[FX]` in mutable references or lists of type `List[Person[FX]]`, this requires all instances to have the same effect implementation. Like in the second alternative, instances of type `Person[out.effect]` are coupled to the lifetime of the corresponding capability `out`. If an implementation depends on more than one effect, it can only be used in the intersection of the corresponding handler regions. Using path-dependent types, our effect-system allows programmers to express this dependency on the type level.
6.5 Discussion: Properties of the Effect System

Our effect system is based on the λ₀₀ calculus by Zhang and Myers (2019). We embed their calculus into the practical programming language Scala that has no full formal specification. It is therefore not in the scope of this thesis to formally prove properties of our library embedding. Nevertheless, in this section we discuss some properties of our embedded effect system and explain under which assumptions we believe them to hold.

6.5.1 Effect Safety

Effect safety is the absence of runtime errors, caused by capabilities being used outside of their defining handlers. Assuming a sound subset of Scala, such as the pDot calculus (Rapoport and Lhoták, 2019), we are confident that our effect system establishes effect safety – though we do not give formal proofs. Adding mutable variables and fields should also not affect our effect system which relies on stable, that is, immutable paths (Odersky and Zenger, 2005b). The same also holds for native exceptions, though unlike JavaEffekt, our library as presented in this chapter is not prepared to interact with native exceptions. A formal treatment is left to future work. In our experience, adding effect types to existing advanced case studies (such as the Scheduler handler in Section 4.10) helped us to discover subtle bugs. The effect system also guided us in the design of the interface for ambient state as presented in Section 6.2.

6.5.2 Effect Subtyping

By marking the set of capabilities in Control as contravariant, we use Scala’s support for subtyping of intersection types to express effect subtyping. This is an important advantage over effect systems that encode effect rows using type-level lists. In those systems, effect subtyping typically has to be implemented manually by performing type-level computation (Kiselyov and Ishii, 2015). In contrast, using intersection types to express the set of effects integrates well with other Scala features like variance annotations and type bounds. Type inference for monotonically growing intersection types is well supported and in consequence, most return types of effectful functions (like drunkFlip) and effect handlers (like maybe) can be omitted.

6.5.3 Effect Polymorphism

We also reuse Scala’s support for type polymorphism to express effect polymorphic functions. Rytz et al. (2012) give an example of an effect polymorphic, higher-order function:

```scala
def mapM[A, B, E](lst: List[A], f: A ⇒ Control[B, E]): Control[B, E]
```

The function mapM is effect polymorphic in the effects E used by function f. The return type of mapM indicates that it potentially calls f in its implementation and so has the same effects as f. The effects E still need to be handled by the caller of mapM. Handler functions like collect and maybe (Section 6.1) are other examples for effect polymorphic functions. Type inference for calling higher-order functions (like mapM) that do not alter the set of effects is well supported. In contrast, nesting multiple handler applications (like collect and maybe) often requires explicit effect annotations.
6. Scala Effekt – Effect Safety through Regions

6.5.4 Effect Parametricity

The example function \( \text{mapM} \) is polymorphic in the effects \( E \). Following Zhang and Myers (2019), we claim that it \textit{should not be possible} for the implementation of \( \text{mapM} \) to (accidentally) handle any concrete effect in \( E \); no matter what \( E \) will be instantiated to at the call site. That is, in the following user program, we should be able to determine \textit{statically} that \( \text{flip} \) is handled by \( \text{collect} \). No implementation of \( \text{mapM} \) should be able to violate this assumption.

\[
\text{collect} \{ \text{amb} \Rightarrow \text{mapM}(\text{List}(1,2,3), n \Rightarrow \text{amb}.\text{flip}()) \}
\]

Zhang and Myers (2019) refer to this \textit{parametricity} of effect polymorphism as \textit{abstraction safety}. Generally speaking, given a type of a function, parametricity of type polymorphism allows us to infer properties of the function’s implementation (Reynolds, 1983; Wadler, 1989). In the case of effect parametricity, we want to statically determine that a given higher-order function \textit{cannot} handle a particular effect. Otherwise, we speak of \textit{accidental handling}. But does effect parametricity hold for our implementation of Effekt? The answer is subtler than in most other implementations of effects and handlers because Effekt is based on capability passing style and multi-prompt delimited continuations. We need to distinguish two aspects of accidental handling.

\textbf{Implementation abstraction} Accidental handling of effects can occur in languages without static effect systems like Eff (Bauer and Pretnar, 2015) and Multicore OCaml (Dolan et al., 2014). Those languages dynamically search handlers at runtime. In those systems, \( \text{mapM} \) could (accidentally or purposefully) handle \( \text{Amb} \) and, for example, change the semantics of \( \text{flip} \) to always return \texttt{true}. Similarly, the presentations of Effekt of Chapters 4 and 5 do not have static effect systems. Still, they already support this aspect of effect parametricity. Capabilities are passed down to their use site and not looked up at runtime. In our example, the function passed to \( \text{mapM} \) closes over the capability \( \text{amb} \), which fixes the implementation of \( \text{flip} \).

\textbf{Control-flow abstraction} Because Effekt uses an implementation of multi-prompt delimited continuations, there is a second aspect of effect parametricity to consider. Looking at the example call to \( \text{mapM} \) again, we would also like to be sure that the continuation captured by \( \text{flip} \) will always be delimited by the corresponding call to \( \text{collect} \) and nowhere else. Because \( \text{mapM} \) does not know about \( \text{amb} \), it should not be possible for \( \text{mapM} \) to delimit the continuation captured by \( \text{amb}.\text{flip}() \). However, as we will see next, in Effekt we can construct examples that violate this property.\footnote{The observation of the loss of parametricity is due to an anonymous reviewer, who we are grateful for. The reviewer also presented a counter-example, which, in adapted form, our analysis in this section builds on.}

\textbf{Capturing the Delimiter by Capturing the Continuation}

The following example uses the untyped \( \lambda \text{dep} \) calculus (Section 2.2) to illustrate (accidental) delimiting of continuations.

\[
def \text{example}() = \text{reset}_{\text{exc}} \{ \text{delimit}(\lambda()). \text{shift}_{\text{exc}} \{ k \Rightarrow \text{"aborted"} \}; \text{"resumed"} \}
\]

In the example, we use \( \text{reset}_{\text{exc}} \) to delimit a program with a prompt \( \text{exc} \). Within the range of \( \text{reset}_{\text{exc}} \), we then use \( \text{shift}_{\text{exc}} \) to abort the current computation and return the string \"aborted\" as the overall result. Assuming that \text{delimit} has no access to the prompt \( \text{exc} \), we might want...
to reason statically, that the continuation captured by \texttt{shift\_exc} is delimited by the surrounding call to \texttt{reset\_exc}. Independent of the implementation of \texttt{delimit}, running the program should always return the string "\texttt{aborted}". This neither holds in \texttt{\lambda\_dcp}, nor in any of the versions of \texttt{Effekt} presented in this thesis. The following code illustrates how to violate this assumption.

Given some prompt \( r \), we can define the function \texttt{delimit}:

\[
\text{def delimit} (\texttt{prog}) = (\texttt{shift} \{ k \Rightarrow k(\lambda(). k(\texttt{prog})) \}()) ;
\]

\[
\text{reset}_r \{ \texttt{example}() \}
\]

The call to \texttt{shift} returns a computation (highlighted in grey). In particular, it returns a computation that, when forced, calls the continuation again with the provided program \texttt{prog}. We finally force this computation outside of the call to \texttt{shift}. Operationally, passing the continuation to the continuation itself duplicates the evaluation context between the call to \texttt{delimit} and (including) the call to \texttt{reset}. Let us recall the operational semantics of \texttt{lambda\_dcp} (Section 2.2).

In particular, rule \(-\text{shift}+\):

\[
(-\text{shift}+) \quad \text{reset}_p \{ \texttt{H}_p[ \texttt{shift} \{ k \Rightarrow e \} ] \} \rightarrow e[k \Rightarrow \lambda x. \text{reset}_p \{ \texttt{H}_p[ x ] \}]
\]

Evaluating the call to \texttt{shift} reifies the capture context \texttt{H}_p and binds it to \( k \). Importantly, the capture context \texttt{H}_p can contain arbitrary other delimiter frames \texttt{reset}_p\{ \[ □ \] \} where \( p \neq p' \).

Calling the continuation reinstall those delimiters. Given the above implementation of \texttt{delimit}, we have the following reduction:

\[
\text{reset}, \{ \texttt{H}_r[ \texttt{delimit} \{ e \} ] \} \rightarrow^* \text{reset}, \{ \texttt{H}_r[ \texttt{reset}, \{ \texttt{H}_r[ e ] \} ] \}
\]

That is, in our example, the capture context contains the delimiter with prompt \texttt{exc}, which is

\[
\texttt{H}_r = \text{reset\_exc} \{ □; "\texttt{resumed}" \}
\]

If \texttt{body} now captures and discards the continuation, only the innermost copy of the context \texttt{H}_r is removed. Hence, our example returns "\texttt{resumed}". To emphasize again, delimiting the extent of the continuation captured by \texttt{shift\_exc} is possible without direct access to the prompt marker \texttt{exc}.

### Violating Control-Flow Abstraction in ScalaEffekt

We can translate the above example to the effect-safe handler setting of ScalaEffekt.

```scala
trait Delimit[R, E] extends Eff {
  def delimited[FX](prog: Control[R, FX]): Control[Control[R, FX & E], effect] =
  delimited(prog) flatMap { c ⇒ c }
}
```

The \texttt{Delimit} effect has a primitive effect operation \texttt{delimited} and a derived effect \texttt{delimit}. The latter simply calls the former and forces the returned computation. The type parameter \( E \) describes the set of effects at the handler—\( FX \) describes the set of effects at the call to \texttt{delimit}.

It is an effect polymorphic effect operation. We implement \texttt{Delimit} like before:

```scala
  def unit(result: R) = pure { result }
  def delimited[FX](prog: Control[R, FX]) = use { k ⇒ k(k(prog)) }
}
```

151
6. Scala Effekt – Effect Safety through Regions

We do not need to thunk the result of \( k(prog) \) since it already is a computation of type Control\([R, FX & E]\). Intuitively, all handlers of effects present in \( FX \), but not present in \( E \), are duplicated by calling delimited. In the following example, we use the effect Delimit together with the exception effect.

```scala
new Reset[Option[String], Pure] handle { r ⇒
    new Maybe handle { exc ⇒
        r.delimit {
            exc.raise("Abort")
        } map { _ ⇒ "Resumed" }
    }
}
```

Like the above program in \( \lambda_{dcp} \), the example returns Some("Resumed"). Importantly, the type of the effect operation delimited does not mention the exception effect. Instead, it is parametric in the effects \( FX \). So why can it handle (or, to be more precise delimited) the exception effect? In our embedding of Effekt, the effect type of the continuation \( k \) does not represent the fact that it captures delimiters. It only mentions the effects it uses and not the ones it delimited. However, the loss of parametricity is not a threat to effect safety. Handlers are duplicated, not removed.

6.5.5 Effect Encapsulation

Another property, effect encapsulation, is related to effect parametricity and can be observed in languages featuring an ML-like type system with row-polymorphism for effect types like Koka and Frank (Lindley, 2018; Leijen, 2018b). The following program is adapted from Leijen (2018b) and written in the Koka language. It shows how an effect used by \( f_1 \) leaks into its type – it is not encapsulated.

```koka
fun f1(action: () → <exc|e> a): e option<a> { // types inferred
    maybe { if (...) { raise("abort") }; action() }
}
```

Here, \( f_1 \) is a higher-order function taking an effectful function \( action \) as argument. The function \( f_1 \) uses exceptions in its implementation but locally handles them with the maybe handler. This implementation detail leaks as part of the inferred type, which states the fact that exc effects of \( action \) will be handled by \( f_1 \). Koka implements effect subtyping via row polymorphism (Leijen, 2014), so the effect row of \( action() \) is unified with those of other statements under maybe.

Manually Encapsulating Effects

Operationally, Koka will handle any exception effect used in \( action \) with the maybe handler in \( f_1 \). We cannot hide this fact by annotating the parameter \( action \) of \( f_1 \) with the type \( () → e a \). This does not type check. However, if we do not want any exceptions thrown by \( action \) to be handled by maybe, languages like Koka and Frank offer some form of manual lifting operation (Biernacki et al., 2017; Convent et al., 2020).

\[\text{This has been brought to our attention by the aforementioned, anonymous reviewer.}\]
Manually injecting the `exc` effect into the effect row also has operational content as described by Leijen (2018b): the runtime search for the exception handler will skip the next handler for `exc`. Now the type of \( f_2 \) truthfully states that it does not handle any effects in \( e \) – including any exception effects.

In \textit{Effekt}, we can also express the two variants of the function \( f \) with different types:

\[
def f_1[A, E](\text{action: } (\text{exc: Exc}) \Rightarrow \text{Control}[A, \text{exc.effect} & E]): \text{Control}[\text{Option}[A], E] 
def f_2[A, E](\text{action: } (): \Rightarrow \text{Control}[A, E]): \text{Control}[\text{Option}[A], E]
\]

The type of \( f_1 \) makes it clear that \text{action} has an unhandled exception effect, handled by \( f_1 \).

### 6.6 Related Work and Chapter Conclusion

In this section, we discuss closely related work. In particular, we compare our approach of achieving effect safety with others.

#### 6.6.1 Abstraction-Safe Effect Handlers via Tunneling

Both, the dynamic and static semantics of \textit{Effekt} is closely related to \( \lambda_{\text{EE}} \), presented by Zhang and Myers (2019). Handling an effect with a handler \( h \) introduces a fresh label. Like prompts in \textit{Effekt}, the label is used on the term level to delimit the scope of captured continuations. As in the first presented versions of \textit{ScalaEffekt} (Brachthäuser and Schuster, 2017), capabilities in \( \lambda_{\text{EE}} \) are tuples of a label and the handler implementation. To ensure effect safety, Zhang and Myers use a simple form of dependent types: Using an effect handler \( h \) introduces \( h会影响到 \text{effect type}, which is effectively a set of labels. The dependent label corresponds to the abstract type member \texttt{h.effect} in \textit{ScalaEffekt}. Like in \textit{ScalaEffekt}, this dependent effect type can only be discharged by the very same delimiter (denoted by \( \ell \)) that introduced the label. Due to the embedding of \textit{Effekt} in Scala, prompts are first class while labels in \( \lambda_{\text{EE}} \) are not first class. Instead, the binding of a label and its use in a handler implementation is statically scoped.

Zhang and Myers formalize \( \lambda_{\text{EE}} \) and show effect safety. However, they do not provide an implementation of their calculus. We use intersection types and path-dependent types to encode the ideas of the \( \lambda_{\text{EE}} \) effect system and thereby make \textit{Effekt} effect-safe.

To facilitate comparison of \( \lambda_{\text{EE}} \) with \textit{Effekt}. Figure 6.8 presents an embedding of the \( \lambda_{\text{EE}} \)-calculus (Zhang and Myers, 2019) into Scala. We use the \texttt{Control} monad to express the operational semantics. We present a practical embedding into Scala and leave a formal translation and corresponding soundness proofs to future work. However, assuming a sound subset of Scala that corresponds to \( \lambda_{\text{EE}} \), we conjecture that our embedding faithfully models the calculus by Zhang and Myers. In particular, the restriction to a subset of Scala excludes the use of mutable state, recursive function definitions, recursive data-types, and exceptions. In addition, effect signatures have to be declared on the top level and should neither use mixin composition, type
6. Scala Effekt – Effect Safety through Regions

```scala
type ℓ[T, E] = Prompt[T, E]

trait F[A, B] { type lbl; def apply(arg: A): Control[B, this.lbl] }

  type lbl = label.type
  def apply(arg: A) = shift(label) { k ⇒ impl(arg)(k) }
}

def δ[R, E](prog: (l: ℓ[R, E]) ⇒ Control[R, l.lbl & E]): Control[R, E] = reset { prog }
def δ[A, B](f: F[A, B]): A ⇒ Control[B, f.lbl] = a ⇒ f.apply(a)
```

Figure 6.8. Using Control to embed the $\lambda_\Omega$ calculus into Scala.

Example To ease comparison with the original calculus, we use $\ell$ as the type of labels and $F$ as the type of effect signatures. Effect signatures $F[A, B]$ only declare a single effect operation with argument type $A$ and return type $B$. For example, we can express the signatures of the ambiguity and exception effect as:

```scala
trait Amb extends F[Unit, Boolean]
trait Exc extends F[String, Nothing]
```

Handlers, modeled by the class `Handler`, are pairs of labels and effect implementations. In this style, the handler for expressions can be expressed as:

```scala
def maybe[R, E](prog: (exc: Exc) ⇒ Control[R, exc.effect & E]) = δ { l ⇒
  val exc = new Handler(l, msg ⇒ k ⇒ pure(None)) with Exc
  prog(exc) map { r ⇒ Some(r) }
}
```

Calling an effect operation amounts to calling $\diamond$ on the handler:

```scala
maybe { h ⇒ ... δ(h)("Failed!") ... }
```

While in the $\lambda_\Omega$ calculus, the operation $\diamond$ performs the continuation capture, in the embedding we perform the capturing in the implementation of `Handler.apply` by means of `shift(l)`. This is necessary to have the available answer types ($T$ and $E$) in scope. We use subtyping (i.e., `Handler <: F`) to existentially hide the answer types and other implementation details of `Handler` when passing capabilities of type $F$ (or `Exc` and `Amb` to be precise).

36Like with the handlers of ScalaEffekt, the field `label` needs to be refined to the singleton type $\ell$.type. That is, the constructor calls requires annotation: `new Handler[String, Nothing, Option[R], E, ℓ.type](...)`. 

154
6.6. Related Work and Chapter Conclusion

6.6.2 Effect Safety by Region Safety

In ScalaEffekt, we establish effect safety by ensuring that a capability cannot be used outside of its handler region. The problem of ensuring region safety is a general one and a wide range of solutions exist.

Safety for multi-prompt delimited control As in our presentation in Section 4.2, Dybvig et al. (2007) guarantee answer type safety by indexing prompts with the expected answer type. Furthermore, they use rank-2 types to prevent prompts from being used across different instances of run. But, as they observe, this is not enough to achieve effect safety, which they explicitly leave to future work. We generalize the idea of region safety and guarantee that capabilities cannot be used outside of the scope that they are created in. Instead of rank-2 types, we use abstract type members. This lets us easily nest scopes using intersection types. As of now, Dotty has better support for path-dependent function types than for rank-2 types. For instance, we can use the lambda syntax (e.g., amb ⇒ amb.flip()), which is currently not possible with rank-2 types.

Safety for effects Many languages with effect handlers base their effect system on some form of row polymorphism. Prominent examples are Koka (Leijen, 2014), Frank (Lindley et al., 2017; Convent et al., 2020), and Links (Hillerström and Lindley, 2016; Hillerström et al., 2017). In contrast, effect-safe library embeddings like Extensible Effects (Kiselyov et al., 2013; Kiselyov and Ishii, 2015) often use various forms of open union types (Swierstra, 2008) to track the list of unhandled effects. In Effekt, we index the monad for effectful computations with an intersection of all capabilities effects used by the computation. This way, capabilities cannot be used outside of their handler region.

Safety for resources To achieve resource safety, Kiselyov and Shan (2008) generalize from a single region (Launchbury and Sabry, 1997; Moggi and Sabry, 2001) to multiple nested regions. They achieve region polymorphism and region subtyping together with good type inference for their library in Haskell. On the type level, they represent nested regions as a nesting of monad transformers. In contrast, we represent nested delimiters by an intersection of abstract type members. To achieve region polymorphism, they reuse Haskell’s polymorphism and to achieve region subtyping they use Haskell’s type class instance search. To achieve effect polymorphism, we reuse Scala’s polymorphism and to achieve subtyping we reuse subtyping for intersection types built into Scala.

Safety for variable scopes Parreaux et al. (2017) apply a strategy very similar to ours in order to implement scope safety in the context of type-safe meta programming. The type parameter Ctx of their type Code[Typ, -Ctx] is used to track the set of free variables. Additionally, variables have a type member that describes the scope the variable can be used in.

```scala
class Variable[A] {
  type Ctx;
  def substitute[T,C](pgm: Code[T, Ctx & C], v: Code[A, C]): Code[T,C]
}
```

As can be seen from the type of substitute, substitution of free variables removes Ctx from the intersection type and thus corresponds to handling of effects.
6. Scala Effekt – Effect Safety through Regions

Safety for capabilities Like in this thesis, Osvald et al. (2016) perform capability passing in Scala: A capability serves as a constructive proof, that the holder is entitled to use the actions associated with the capability. To prevent leaking of capabilities, Osvald et al. (2016) introduce a type-based escape analysis as an alternative approach to traditional effect systems: arguments to functions can be marked as second class (or “local”). The type checker then guarantees that the capability cannot leave the dynamic scope of the function call. Liu (2016) presents a different approach to capability based effect safety, by distinguishing between functions that can capture capabilities and others that cannot (called “stoic”). In our design of Effekt, we adopt the capability passing of Osvald et al. (2016). Their approach of second class values might be an interesting alternative to the effect system presented in this thesis. However, they are not available in Scala, while our library can readily be used.

6.6.3 Effect Parametricity

Our implementation of ScalaEffekt does not guarantee effect parametricity due to accidental delimiter capture (Section 6.5). Effect polymorphic effect operations like delimited may seem contrived. However, effect polymorphic effect operations are essential to express examples like Async.async (Figure 6.7a). Languages without support for effect polymorphic operations, like \( \lambda \) or Koka cannot directly express this signature. To work around this limitation in Koka, Leijen (2017a) fixes the set of effects used by asynchronous programs to io. We embed Effekt into Scala and use intersection types as sets of effects. This allows us to reuse Scala’s type system for effects. However, as shown in Section 6.5, it also allows to violate control-flow abstraction. We leave it to future work to estimate the impact of lost parametricity and to implement strategies to restore it. We see two ways to restore parametricity in ScalaEffekt – both come with significant trade-offs.

Restricting effect signatures We conjecture that control-flow abstraction can be restored by restricting the expressivity of effect signatures. Languages like Koka or the \( \lambda \mathrm{oo} \) support effect polymorphic effect operations and thus cannot express operations like delimited (Section 6.6.1). In Effekt, effect signatures are arbitrary traits and effect operations can use Scala’s polymorphism to express effect polymorphism (e.g., type parameter FX in the signature of delimited). This is not expressible in our embedding of \( \lambda \mathrm{oo} \) (Figure 6.8). There, one effect signature \( F[A, B] \) always only describes a single effect operation from type \( A \) to \( B \), which excludes effect polymorphic operations. While this may restore effect parametricity, this might exclude many of the newly supported extensibility scenarios of Section 4.4.

Resuming continuations as effect To duplicate parts of the evaluation context and the delimiters contained therein, the implementation of the effect operation delimit calls the continuation with the continuation. It thus relies on the fact that the continuation can escape the scope of the shift that captured it! We can prevent this by treating continuation resumption itself as an effect:

```scala
```
This way, the continuation can only be used within the scope of the body provided to \texttt{shift}. While we believe that this solution restores parametricity, it also rules out handlers (\textit{e.g.}, \texttt{Scheduler}, Figure 4.9), which need to store the continuation to resume it later.

6.6.4 Effect Handlers and Object-Oriented Programming

In Chapters 4 and 5, we started to explore the combination of effect handlers and object orientation. However, those versions of \texttt{Effekt} did not guarantee effect safety. The present chapter shows how to add effect safety to \texttt{Effekt} and support effect polymorphism. As highlighted in Section 4.7.3, effect-safe programming with effect handlers in a language with objects comes with new challenges – mediating encapsulation and flexible use of objects. Inostroza and van der Storm (2018) also combine effect handlers and object orientation in the language JEff. In JEff, the continuation takes an updated copy of the effect handler as additional argument, which models dynamically scoped state (Kiselyov et al., 2006). It also allows to change the handler implementation for the rest of the computation, similar to shallow handlers. The effect system of JEff does not feature effect polymorphism and hence problems with effect encapsulation do not arise. Inostroza and van der Storm (2018) present a static effect system for the language JEff. However, in contrast to our embedded effect system, JEff does not support effect polymorphism. In the simpler setting without effect polymorphic functions, violations of effect parametricity are not a concern.

6.6.5 Conclusion

In this chapter, we presented \texttt{Effekt}, a monadic library for programming with effect handlers in Scala that features effect polymorphism, effect subtyping and effect safety. We use intersection types and path-dependent types to track the set of effects a program might use. This allows us to directly reuse Scala’s support for polymorphism for effect polymorphism and Scala’s support for subtyping for effect subtyping. Combining effect-safe delimited control with object-oriented programming both offers new ways to modularize effectful programs but also comes with new challenges.
Chapter 7
Discussion and Conclusion

Effect handlers enable control-flow abstractions that are user-definable, modular, and compositional. In this thesis, we explored the modularity and extensibility opportunities gained by combining effect handlers with object-oriented programming. First, we revised two important aspects of effect handlers, delimited control (Chapter 2) and dynamic binding (Chapter 3) to elaborate the operational and conceptual essence of effect handlers. Building on this foundation, we presented a design to integrate effect handlers with object-oriented programming (Chapter 4). We showed how to enable user programs to be written in direct-style (Chapter 5) and how to establish effect safety (Chapter 6). In this chapter, we summarize the conclusions of the previous chapters, highlight the most important aspects of each chapter, discuss some limitations, and sketch potential future work.

**Delimited control** In the first part of this thesis, we revisited several different forms of delimited control in the literature. We presented existing generalizations of delimited control in two steps to arrive at effect handlers. These two steps helped us to highlight important characteristics of effect handlers: First, the generalization from one control operator to a family of control operators enables the safe use of multiple different effects in one program. With multi-prompt delimited control, prompt markers can be used to uniquely describe the connection between a control operator \( \text{shift}_p \) and its corresponding delimiter \( \text{reset}_p \) without having to risk interference with other operators and delimiters. Second, syntactically moving the implementation of a control effect from the control operator to the delimiter simplifies reasoning about the continuation usage. Maybe most importantly, it emphasizes that handlers perform effect encapsulation: Handlers can use other control effects to implement effect operations. However, these control effects are not evaluated at the call site of the implemented effect operation – they are evaluated at the definition site of the handler. Effect handlers thus offer an interesting combination of dynamic and static scoping. Handlers dynamically bind effect operations, but execute their implementation in the static scope of their definition.

**Dynamic binding** Starting from dynamic binding (e.g., ambient values), we again performed two steps of generalization to arrive at effect handlers. While the generalizations from delimited control carried some insight, but no technical novelty – this is different for the generalization from dynamic binding. In particular, the novel feature of ambient functions provides the same aforementioned effect encapsulation that effect handlers provide, without the complexity of delimited continuations. Avoiding this complexity bears fruit for both language implementors and users. On the one hand, it is easier to efficiently implement ambient functions than it is to implement ambient control. This is the case since ambient functions are always tail-resumptive, which allows important optimizations (Leijen, 2017b). On the other hand, users can readily apply their intuition about closure and lexical scoping to reason about effect encapsulation. In the future, we hope to see more languages that will support this feature.
7. Discussion and Conclusion

**Effekt** In the second part of this thesis, we presented Effekt, a library design for effect handlers that integrates well with object-oriented programming languages. Building on the insights from the first part, we designed Effekt around the concept of explicit capability-passing style as an alternative to dynamic binding. Explicit capability-passing style has several advantages. Since it only requires basic features, it is well supported by existing programming languages. Modern implementations of the Java Virtual Machine offer a multitude of optimization strategies for dynamic dispatch and virtual method calls such as polymorphic inline caches (Hölzle et al., 1991) and inlining of monomorphic calls (Kotzmann et al., 2008). Explicit capability passing directly benefits from these optimizations. Combining capability passing with multi-prompt delimited control also gives rise to an alternative of expressing effect polymorphism (Osvald et al., 2016). Lastly, capability passing helps us to understand effects as imposing an additional requirement on the calling context, rather than a side effect that occurs in addition to computing the result. In our effect-safe implementation, we used this insight and marked the type parameter Effect on Control as contravariant.

The combination of effect handlers and object-oriented programming proved to be a fruitful one. On the one hand, embedding effect handlers in an object-oriented programming offers new ways to modularize effect handlers. On the other hand, having control effects and handlers available in a language allows users to express user-defined control-flow constructs. Nevertheless, the combination also comes with new challenges. If no special measures are taken, allocation of objects and modification of fields has to be understood as global effects. In contrast, effect handlers express control flow locally. They can use the captured continuation in non-linear ways, potentially violating assumptions about global resources, such as heap-allocated objects. In Section 4.3 we presented a specialized state handler to implement ambient state, which interacts well with non-linear control flow. In particular, it implements the desired backtracking behavior, when resuming a continuation multiple times. The solution of handlers with ambient state is only a partial one as it only backtracks the handler state shallowly. In future work, it would be interesting to develop a more holistic approach to integrate heap-state with effect handlers – potentially following Leijen (2018b).

**Direct-style Effekt in Java** We presented the first library design for programming with effect handlers in Java. In contrast to our Scala implementations, programs in JavaEffekt can be written in direct style. We enable the direct style by applying a CPS transformation on the level of bytecode. Our transformation reduces stack utilization by trampolining, performs resumptions in constant time (in the stack-depth), supports multiple resumptions and is competitive in its performance. In the future, once delimited continuations are natively supported by the JVM (Pressler, 2017), the bytecode transformation backend could be switched to an implementation in terms of native continuations. As of this writing, switching to the development version of the native implementation still comes with performance penalties. More importantly, it does not (yet) allow multiple resumptions.

**Effect-safe Effekt in Scala** To the best of our knowledge, we are the first to present an effect-safe embedding of monadic, multi-prompt delimited control. Our effect-system is conservative and excludes some programs that could be expressed safely without effect-types (Kobori et al., 2016). Building on monadic delimited control, we then showed how to make our Scala implementation of Effekt effect safe. We use intersection types and path-dependent types to track the set of effects a program might use. This allows us to readily reuse Scala’s support for polymorphism to express effect polymorphism and Scala’s support for subtyping to express effect subtyping. Our
7.1 Future Work: Effect-Safe and Direct-Style Effekt

We identified two major limitations of our Effekt implementation in Chapter 4: user programs have to be written in monadic style and the implementation does not guarantee effect safety. We separately addressed the two issues in Chapters 5 and 6.

In the future, it would be interesting to combine the two solutions and obtain an effect-safe embedding where user programs can be written in direct-style. However, Java's type system is not expressive enough to embed an effect system in the style of Chapter 6. In particular, it does not support path-dependent types. We conjecture, the most promising way is to use the effect system of ScalaEffekt together with a CPS translation like the one presented in Chapter 5.

Since Scala does not support checked exceptions, one could instead use annotations to guide the transformation (Rompf et al., 2009; Parallel Universe Software Co., 2013). Rompf et al. (2009) implement delimited continuations as a Scala compiler plugin. They annotate effectful programs with \( A @\text{cps}[B, C] \) denoting a computation that computes a result of type \( A \) and changes the answer type from \( B \) to \( C \) (Rompf et al., 2009). Since annotations are not automatically propagated (as opposed to checked exceptions in Java) Rompf et al. need to manually implement annotation inference and checking of annotations. We could employ a similar strategy and instead of the monadic type constructor \( \text{Control}[+A, -\text{Effects}] \) use an annotation \( A @\text{effects}[-\text{Effects}] \). Using such an annotation based approach, our running example could be expressed as

```scala
def drunkFlip(amb: Amb, exc: Exc): String @effects[amb.effect & exc.effect] =
  if (amb.flip()) {
    if (amb.flip()) { "Heads" } else { "Tails" }
  } else {
    exc.raise("Too drunk")
  }
```

Like the type constructor \( \text{Control} \), the effects annotation could be indexed by a contravariant intersection type to track the set of used effects.
7. Discussion and Conclusion

7.2 Future Work: Efficient Compilation of Effect Handlers

Effect handlers allow high-level, user-definable, and composable control abstractions. However, a significant runtime cost is associated with searching the correct handlers and capturing the continuation. By means of capability passing, our implementations of Effekt reduce the handler search to a simple dynamic dispatch. Still, capturing the continuation involves a runtime search through the (user level) stack. This becomes visible in our performance evaluation of Section 5.6.2. Comparing the baseline, not using effects, with our optimized version of JavaEffekt suggests room for improvement of around 27x for the Stateloop benchmark and 7x for the NQueens benchmark. Future work could investigate how to further improve the performance of our effect handler implementations. There exist two lines of closely related work to minimize the performance overhead of the abstraction of effect handlers: runtime optimizations and compile-time optimizations.

Runtime optimizations

Some languages limit the expressivity of effect handlers to allow more efficient implementation strategies in the language runtime. For example, Multicore OCaml originally only allowed continuations to be called once to support efficient (and destructive) stack switching (Dolan et al., 2014, 2013). Other languages detect usage patterns of the continuation, which then can be supported more efficiently. In the Koka implementation, Leijen (2017b) syntactically recognizes that the continuation is only used once in tail position to optimize capture / resume sequences.

Compile-time optimizations

Leijen (2017c) uses effect types to distinguish pure from effectful computations and only applies a CPS transformation to the latter. Pretnar et al. (2017) explore compile time source-to-source transformation rules to implement an optimizing compiler for the language Eff. The idea is to repeatedly apply rewrite rules in order to specialize effectful programs to their handlers (Pretnar et al., 2017).

In previous work (Schuster and Brachthäuser, 2018), we explored an alternative strategy to efficiently compile control effects. We started to adapt the approach as a compilation strategy for effect handlers (Schuster et al., 2020), which we outline in the remainder of this subsection. Our compilation strategy rests on the following observations: As pointed out in Chapter 2, the semantics of a program with control-effects depends on the evaluation context (the stack). The concrete stack can only be known at runtime. But, what if certain information about the stack can be determined statically at compile time? In languages like Koka (Leijen, 2017c), the stack carries both, the effect handler implementations as well as markers delimiting the captured continuations. Both are part of the evaluation context, as discussed in Chapter 2. If some of this information would be available at compile-time, can we use it to partially evaluate (that is, specialize) the program? We distinguish the following two classes of potentially static information:

Stack shape.

We might know in which order effect handlers will be present on the stack, when evaluating a part of the program. We refer to this information as the shape of the stack. Statically knowing the order in which effect handlers appear would allow us to specialize the control flow and continuation capture. For instance, we could use this information to drive an iterated CPS translation (Danvy and Filinski, 1990), that is, perform one CPS translation for every
handler in the stack shape. This is similar to the translation by Hillerström et al. (2017), however, we propose to additionally also take the static ordering of effect handlers into account. Like in the CPS hierarchy by Danvy and Filinski (1990), dynamic runtime search for a delimiter could be replaced by direct composition of explicit continuation fragments.

**Handler implementation.**

We might know the concrete handler implementation that will be present on the stack when evaluating a particular part of the program. This would allow inlining of the handler implementations at the call site of the effect operation. Inlining the handlers not only removes the runtime search for the handler implementation, but also potentially opens up further local optimizations.

We believe that capability-passing style, as performed by Effekt, can be an important step to achieve handler inlining. Use staging annotations (Taha and Sheard, 1997) capabilities can be marked as stage-time information. This is similar to how Kammar et al. (2013) use Haskell’s support for inlining type-class dictionaries. In future work, it would be interesting to fully develop the ideas into a compiler for a language with effect handlers like Koka. This will come with interesting challenges, since in Koka, both the stack shape and handlers are not always statically known. However, we believe that it is beneficial to start from the restricting assumption that handlers and the stack shape are always statically known and guarantee full elimination of the associated runtime costs. We conjecture that having to lift some of the restrictions that come with this requirement will make the associated runtime costs explicit.

### 7.3 Future Work: Effectful Traversals and Modular Interpreters

Often, interpreters for programming languages are structured as recursive functions over the abstract syntax tree of the object language. If the recursion structure of the interpreter follows the recursion structure of the object language, the traversal itself can be defined separately as a fold. Maybe most importantly, separating the recursion structure from the computation immediately gives rise to fusion of traversals (Meijer et al., 1991).

Often, interpreters for programming languages use computational effects of the host language to express effects of the object language. Another way to model effects of the object language is by monads (Wadler, 1995). Expressing effects with monads allows implementing interpreters in host languages that do not support the effects of the object language.

The two approaches can be naturally combined to implement interpreters using effectful (or monadic) folds (Meijer and Jeuring, 1995). This allows describing traversals that can be fused, while being able to use effects not available in the host language. Object algebras (Oliveira and Cook, 2012; Oliveira et al., 2013) are a recent way to describe folds in a modular and extensible way. Different aspects of a traversal can be implemented in separate modules, while still allowing some dependencies. Computations can access attributes computed on the current node and on immediate child nodes. The following example, adopted from Oliveira et al. (2013), shows a simple algebra for arithmetic expressions with numeric literals and addition:

```scala
trait ExpAlg[E] {
  def lit(n: Int): E
  def add(e1: E, e2: E): E
}
```
7. Discussion and Conclusion

Instead of constructing the recursive data structure, terms are represented by their church encoding (i.e., \( \forall r. (\text{ExpAlg}[r] \Rightarrow r) \Rightarrow r \)) and are constructed by immediately calling into the algebra (Barendregt, 1992; Oliveira and Cook, 2012).

In future work, it would be interesting to extend object algebras to support modular effectful traversals. We believe that combining object algebras with effect handlers as presented in this thesis is especially promising. Different components of effectful traversals can use different monads with different semantic domains, which might be difficult to compose. In contrast, using \texttt{Effekt} the only necessary monad is \texttt{Control} and effectful algebras can easily be specialized to it:

\[
type 
\text{ExpEff}[E] = \text{ExpAlg}[\text{Control}[E]]
\]

Inostroza and Storm (2015) extend the object algebra approach to modularly propagate context information (like a type-environment) through the traversal. Each traversal component can depend on different aspects of the context. The framework of Inostroza and Storm (2015) hides the necessary context composition and projections. The same approach could potentially be used to propagate capabilities.

In prior work (Rendel et al., 2014), we extended the expressiveness of object algebras to also allow dependencies on left siblings, encoding the full class of L-attributed grammars. This way, object algebras can be used to modularly implement one-pass compilers. Using the results of this thesis, traversals expressed with object algebras can also be effectful. That is, attribute specifications can use control effects. Using the encoding techniques of Rendel et al. (2014) together with the design for effect handlers, as presented in this thesis, it might be possible to implement effectful one-pass compilers.

Combining tree traversals with delimited control effects is particularly interesting, since it allows expressing non-local tree rewritings, such as let-insertion (Yallop, 2016, 2017) or transformations to A-normal form (Thiemann, 1996). Similarly, it has recently been shown by Wang et al. (2019) that delimited control operators can be used to express reverse-mode automatic differentiation. Using effectful tree traversals, it might be possible to describe such transformations as separate, reusable modules.

7.4 Future Work: Naturalistic DSLs and Effectful Syntax

It is the goal of domain specific languages (DSLs) to bridge the conceptual gap between languages used in the particular domain and the computer language domain experts use to solve the domain problems (Hudak, 1996). Domain language is often close to natural (i.e., spoken) language, making the task of DSL design particularly challenging (Lopes et al., 2003). Natural language has the reputation of being lexically and syntactically ambiguous, having complicated and context dependent binding structures, and often has non-trivial semantics, which rarely is compositional.

In consequence, often DSLs are still far away from being close to natural language. This is in particular the case for DSLs, which are embedded (Hudak, 1996) into a general-purpose language. With embedded DSLs, the host language additionally imposes its own syntactical restrictions and typing discipline on the DSL designer. Many limitations have been addressed in their own line of work. Examples include syntax extensions as libraries (Erdweg et al., 2011; Biboudis et al., 2016) and domain specific type system extensions (Lorenzen and Erdweg, 2016)). However, non-context-free linguistic constructs are often neglected.

In about the last decade, many developments in modeling the semantics of natural languages
have been inspired by computer science and the theory of abstract machines and control operators in particular. Importantly, Shan (2005) describes “noncompositional phenomena in natural languages” as *linguistic side effects*. Delimited continuations have successfully been used to model linguistic side effects, such as quantification, focus, and polymorphic coordination (Shan, 2004a, 2005; Barker and Shan, 2004). Maršík and Amblard (2016) recently used algebraic effects with handlers to give a compositional semantics to deixis (“John loves me”), quantification with scope islands (“John loves every woman”), and implicature (“John, my best friend, loves me”).

Implementors of (domain specific) programming languages often reside to effects to describe the *semantics* of the language (Wadler, 1995). Inspired by the recent developments in natural language semantics, we propose to follow Maršík and Amblard (2016) and use effects and handlers to describe the *syntax* of a DSL. In particular, we propose to group the different syntactic constructs of a DSL according to the following aspects: (1) pure syntax, that can be understood as compositional construction of the abstract syntax tree. (2) effectful syntax, that, like linguistic side effects (Shan, 2005), (often) requires context for interpretation and results in some form of non-local rewriting of the syntax tree. Effect operations can be used to express effectful syntax. (3) binding syntax, which provides the necessary context. Binding syntax can be expressed as effect handlers for effectful syntax.

### 7.4.1 Effectful Syntax in Scala

Scala comes equipped with many features that allow designing elegant embedded DSLs (Moors et al., 2012). Examples include infix notation of method application, implicit coercions, implicit resolution, and techniques to extend classes retroactively with new methods (Odersky et al., 2006). Using the results of this thesis, we can now additionally use effect handlers to implement effectful DSLs. To illustrate the gained expressivity, we implement examples from Maršík and Amblard (2016) as an embedded domain specific language in Scala. Maršík and Amblard (2016) already use a calculus of effects and handlers to express the semantics. We simply translate the examples to Scala, using ScalaEffekt as presented in Chapter 6. We also use Scala implicits to hide the details of capability passing (Section 4.6.1) and to focus on the application domain of natural language.

**The Speaker Effect.** We begin with a simple sentence that uses the speaker effect to refer to the contextual speaker of the sentence:

```scala
def s1 = given Speaker = John said { Mary loves me }
```

The declaration of sentence $s_1$ requires an implicit parameter of type `Speaker`, which tells us that the sentence uses the speaker effect:

```scala
trait Speaker extends Eff { def me(): Control[NominalPhrase, effect] }
```

Here, the type `NominalPhrase` is part of the abstract syntax tree of our DSL. Making capability passing and other DSL implementation techniques explicit, the above example corresponds to:

```scala
def s1(s: Speaker) = s.me().map { p ⇒ Said(John, Loves(Mary, p)) }
```

The syntax tree describing the sentence can only be fully constructed, once the speaker is known. The speaker effect can be handled locally by using the handler `saidQuote`:
7. Discussion and Conclusion

    def s_2 = John saidQuote { Mary loves me }

Again, making capability passing and other DSL implementation techniques explicit, the sentence corresponds to:

    def s_2: Control[Sentence, Pure] = saidQuote(John) { s ⇒
        s.me().map { p ⇒ Said(John, Loves(Mary, p)) } }

The type of the sentence now suggests that no effect is left to be handled, and we can run the sentence to obtain Said(John, Loves(Mary, John)).

The Scope effect. Passing down context information, as we did with the speaker effect, does not require full handlers. In Koka we could use the feature of ambient values as presented in Chapter 3. Likewise, in Scala, implicit parameters suffice to express this effect. Things become more interesting when we consider the scope effect, which can be used to model universal quantification (Maršík and Amblard, 2016).

    def s_3 = scoped { John saidQuote { every(Woman) loves me } }

The effect operation every takes a predicate (i.e., Woman) and introduces a universal quantification at the position of the handler scoped. It is declared in the signature of the Scope effect.

    trait Scope extends Eff {
        def every(pred: NominalPhrase ⇒ Sentence): Control[NominalPhrase, effect]
    }

Running s_3, we see that this leads to a systematic “rewrite” of the syntax tree, moving the introduced binder and the predicate up to the handler:

    ▶ Forall(x ⇒ Implies(Woman(x), Said(John, Loves(x, John))))

Every invocation of the effect operation every introduces an additional quantifier. This non-local rewriting of the syntax tree to introduce a binder is very similar to let-insertion (Yallop, 2016, 2017). Yallop (2017) show how to use effect handlers to perform let-insertion.

With effectful syntax, we propose to revisit linguistic side effects (Shan, 2005) in the context of domain specific languages. In particular, we propose to apply Maršík and Amblard’s (2016) idea of using effect handlers to model natural language semantics to embedded domain specific languages. The library presented in this thesis enabled us to combine the expressivity of Scala with the expressivity of effect handlers to modularly implement effectful syntax. We conjecture that systematically using effect handlers and effectful syntax leads to user programs that communicate the usage of linguistic features in their types, opens up new modularization strategies for DSL implementations, and potentially offers improved error reporting and better IDE support. We believe that effectful syntax opens up a new interesting perspective on the design and implementation of naturalistic DSLs. In future work, it would be interesting to explore the idea of effectful syntax on application domains like (financial) contracts (Peyton Jones et al., 2000) or test specifications.
7.5 Conclusion

In this thesis, we addressed our goal to make the powerful program structuring technique of effect handlers available to a wider audience in two ways.

Firstly, we provided a fresh perspective on effect handlers, viewing them as a combination of delimited control and dynamic binding. Approaching effect handlers from dynamic binding, the new feature of ambient functions emerged. Ambient functions are easier to understand and to reason about than effect handlers since they do not manipulate the control flow. At the same time, they show the same powerful effect encapsulation properties of effect handlers. The way to approach handlers from dynamic binding and ambient functions as an intermediate form of abstraction can make a significant difference in a wider adoption of effect handlers.

Secondly, we explored the design space of embedding effect handlers in existing mainstream object-oriented programming languages. We have demonstrated that effect handlers can be implemented efficiently on top of existing object-oriented languages. Our approach finally enables programmers of object-oriented languages to also use effect handlers to structure their programs. At the same time, it allows to use object-oriented programming techniques to structure effect handlers. This way, our language design opens up previously unexplored dimensions of extensibility. We presented a novel bytecode transformation, which enables writing user programs of effect handlers in direct-style. The transformation makes significant use of closure creation and is competitive in performance. Finally, we demonstrated a new way to achieve effect safety in a library embedding of effect handlers. Using intersection types and path-dependent types allowed us to model regions and establish that capabilities cannot be used outside of their region. We are confident that our library design can guide the research on effect handlers towards more modular and extensible language designs, regardless of the programming paradigm.
References


References


References


References


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., Boston, Massachusetts, USA, 1995.


References

Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In Proceedings of the Workshop on Type-Driven Development, New York, NY, USA, 2016. ACM.


References


Oleg Kiselyov. Incremental, undoable parsing in OCaml as the general parser inversion. Posted on the OCaml mailing list, July 2007.


References


References


References


References


179
References


