Compiling Lexical Effect Handlers with Capabilities, Continuations, and Evidence

Dissertation

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

vorgelegt von
Philipp Schuster, M.Sc.
aus Simmern

Tübingen
2022

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	23.01.2023
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Klaus Ostermann
2. Berichterstatter:	Prof. Dr. Peter Thiemann

Danksagung

Natürlich ist es nicht möglich, eine Doktorarbeit ohne Unterstützung zu schreiben. Deshalb möchte ich hier einigen Personen namentlich danken.

Zuallererst danke ich meinem Doktorvater Klaus Ostermann, der mir nicht nur die Möglichkeit gegeben hat in einer großartigen Arbeitsgruppe an einer großartigen Universität zu lernen und zu arbeiten, sondern mir auch die Freiheit geboten hat die Themen zu verfolgen, die mir zusagen. Dann allen meinen Kollegen, von denen ich Paolo, Tillmann, Julian, David und Ingo hervorhebe, weil sie stets gute Diskussionspartner waren. Und auch den Sekretärinnen Birgit und Bettina, die mich weitestgehend vor Bürokratie beschützt haben, und Renate, die es geschafft hat, Prozesse sehr leicht zu halten. Schließlich meinem früheren Mentor Ralf Lämmel, der mich sowohl auf eine akademische Laufbahn, als auch in das Feld der Programmiersprachen gebracht hat.

Als Nächstes danke ich meinem Freund und Vorbild Jona, von dem ich sowohl beruflich als auch persönlich unglaublich viel gelernt habe. Alex, mit dem ich eine gute Zeit in Tübingen hatte, fast so wie in Koblenz. Und Vadim, der spontan auch für ungewöhnliche Aktivitäten zu haben ist.

Zu guter Letzt danke ich meiner Familie, die immer an mich geglaubt hat. Insbesondere meinem Vater, der mir mein Leben lang Unterstützung auf jeder Ebene hat zukommen lassen. Und meinem Bruder, der immer für mich da ist, wenn ich ihn brauche.

Abstract

The ever increasing demand for software in an ever increasing number of different domains leads to an ever increasing complexity of programs. To scale the size and complexity of software, programmers compose programs out of individual, reusable parts. These parts are discovered to be commonly useful and shared as libraries. To enable this sharing and reuse, programming languages must offer a way to abstract over patterns of code and to concretize the pattern to specific use cases. Moreover, as the scale of software increases, it becomes more and more important that programmers are able to ascertain themselves of the correctness of one part of the program without looking at all the other parts.

Effects are a class of language features that makes programs interact with their context in a non-trivial way. Examples are file system access, mutable state, exceptions, generators, and more. Most languages include some of these features in one form or another, which speaks to their usefulness. However, it is widely agreed upon that undisciplined use of effects leads to programs that are hard to get correct, hard to understand, and hard to maintain. Moreover, each of these individual features must be built into a programming language and while doing so their combination must be carefully considered. Finally, programmers cannot abstract over these features and therefore not share them nor extend them in ways unforseen by the programming language implementor.

Effect handlers are a relatively recent programming language feature that subsumes many existing language features for effects, and even goes beyond what most programming languages offer. With effect handlers, effects like mutable state, exceptions, or generators are user-defined. This allows for abstraction over repeated patterns involving different kinds of effects. Naturally, programmers can share and reuse these abstractions in different contexts and even invent their own effects which opens up a whole new design space. Moreover, effect handlers delimit the extent of effects, which gives programmers certain guarantees without inspecting the entire program. In other words, they enforce a disciplined use of effects.

In this thesis we present a compilation technique for effect handlers, which will help this programming language feature to move from theory to practice. Compiled programs do not need any runtime support, which makes it widely *deployable.* It enables aggressive compile-time optimizations, which makes it *efficient.* It supports programs using more exotic effects like non-determinism, which makes it *general.* It is also of theoretical interest, because it targets well-known and well-studied languages.

In summary, our compilation technique enables new kinds of abstractions without regret for programs using effect handlers.

Zusammenfassung

Der ständig steigende Bedarf an Software in immer unterschiedlicheren Bereichen führt zu einer immer größeren Komplexität der Programme. Um die Größe und Komplexität von Software zu skalieren, setzen Programmierer die Programme aus einzelnen, wiederverwendbaren Teilen zusammen. Diese Teile werden als allgemein nützlich erkannt und als Software-Bibliotheken geteilt. Um diese gemeinsame Nutzung und Wiederverwendung zu ermöglichen, müssen Programmiersprachen Möglichkeiten bieten, über Code-Muster zu abstrahieren und diese später wieder zu konkretisieren. Außerdem wird es mit zunehmendem Umfang der Software immer wichtiger, dass die Programmierer in der Lage sind, sich von der Korrektheit eines Teils des Programms zu überzeugen, ohne sich alle anderen Teile anzuschauen.

Effekte sind eine Klasse von Sprachkonstrukten, die Programme auf nicht-triviale Weise mit ihrem Kontext interagieren lässt. Beispiele sind der Zugriff auf das Dateisystem, veränderliche Referenzen, Ausnahmen, Generatoren und mehr. Die meisten Sprachen enthalten eine Auswahl dieser Konstrukte in unterschiedlichen Variationen, was für ihre Nützlichkeit spricht. Es besteht jedoch weitgehend Einigkeit darüber, dass undisziplinierte Verwendung von Effekten zu Programmen führt, die schwer korrekt zu erstellen, schwer zu verstehen und schwer zu pflegen sind. Des Weiteren muss jedes einzelne dieser Konstrukte in jede einzelne Programmiersprache eingebaut werden, und dabei muss ihre Kombination sorgfältig bedacht werden. Schließlich können Programmierer nicht über diese Konstrukte selbst abstrahieren und sie daher nicht gemeinsam nutzen oder in einer Weise erweitern, die der Implementierer der Programmiersprache nicht vorhergesehen hat.

Effekt-Handler sind ein relativ neues Sprachkonstrukt, das viele bestehende Konstrukte für Effekte ausdrücken kann und sogar über das hinausgeht, was die meisten Programmiersprachen anbieten. Mit Effekt-Handlern können Effekte wie veränderliche Referenzen, Ausnahmen oder Generatoren von Benutzern der Programmiersprache definiert werden. Dies ermöglicht die Abstraktion über wiederholte Muster, die Effekte verwenden. Natürlich können die Programmierer diese Abstraktionen in verschiedenen Kontexten gemeinsam nutzen und wiederverwenden und sogar ihre eigenen Effekte erfinden, was ganz neue Gestaltungsmöglichkeiten eröffnet. Darüber hinaus grenzen Effekt-Handler den Kontext von Effekten ab, was dem Programmierer gewisse Garantien gibt, ohne dass er das gesamte Programm inspizieren muss. Mit anderen Worten, sie erzwingen einen disziplinierten Gebrauch von Effekten.

In dieser Thesis stellen wir eine Kompilierungstechnik für Effekt-Handler vor, welche dabei hilft, dieses Sprachkonstrukt von der Theorie in die Praxis zu bringen. Kompilierte Programme benötigen keine Laufzeitunterstützung, was sie weithin einsetzbar macht. Sie ermöglicht aggressive Optimierungen zur Kompilierzeit, was sie effizient macht. Sie unterstützt Programme, die exotischere Effekte wie Nicht-Determinismus verwenden, was sie allgemein einsetzbar macht. Sie ist auch von theoretischem Interesse, da sie in bekannte und gut untersuchte Programmiersprachen übersetzt. Zusammenfassend: Unsere Kompilierungstechnik ermöglicht neue Arten von Abstraktion ohne Bedauern in Programmen, die Effekt-Handler verwenden.

Contents

List of Figures

1 Introduction

Computation is interaction between a program and its context. *Pure programs* interact with their context in a trivial way: they return a value when they are done. Their meaning is independent of their context. This observation directly manifests in the semantics of pure programming languages as a congruence rule. *Effectful programs*, in contrast to pure programs, depend on or modify the context they run in [Wright and Felleisen, 1994]. In other words, the interaction between effectful programs and their contexts is much richer. Examples of language features that make programs effectful are exceptions, mutable state, and filesystem access. Throwing an exception discards a part of the context, mutable state modifies the content of a reference cell, and filesystem access affects the user's hard drive.

From a programmer's point of view, reasoning about effectful programs is more difficult, because their context must be taken into account. It is possible to make the context, which is usually left implicit, explicit. For example, instead of using exceptions, programs can match on error values. Instead of using mutable state, programs can pass along the current value. However, some programs are more understandable when things that are clear from the context are left implicit. Spelling out every little detail is cumbersome. Moreover, interaction with the outside world cannot be emulated like this. Therefore, completely banning all effects is impractical as programs must interact with the world in a non-trivial way.

Effect handlers [Plotkin and Pretnar, 2009, 2013] are an attractive language feature for working with a broad range of effects. With effect handlers, programs can be locally effectful, but handlers delimit the interaction between the program and its context. In a language with effect handlers, effectful programs use abstract effect operations and effect handlers locally provide meaning to them. Handlers delimit the extent of the context that has to be taken into account when reasoning about effectful programs.

For example, a programmer might want to locally use an exception to structure control flow, but also be sure that the exception is always caught and never crashes the program. Or she might want to locally use mutable state in an algorithm, but also be sure that intermediate states are never visible to other parts of the program. File system effects are always global as they affect the context outside of the program, but it might still be useful for a programmer to know if a part of a program might affect the file system or not.

Effect systems [Nielson et al., 1999] track not only the types of values, but also the effects of programs. It is useful to know which effects a program does and does not have. In languages with effect handlers, effect systems enforce the locality of effects. They extend the guarantees of programming languages from type safety to effect safety: all effects are delimited by a corresponding handler. In other words, the effects cannot affect the context outside of the handler. From the outside the program is pure.

For example, for exceptions effect safety means that all exceptions are caught and do not propagate to the user of the program. As another example, a function may internally use mutable state but it is guaranteed that it behaves like a pure function which means local reasoning applies to its uses. Finally, all global effects, like file system access, show up in the signature of the entry point of the program.

1.1 Effect Handlers by Example

The following examples make these intutitions concrete. They are written in Effekt, a language with lexical effect handlers which will be the starting point of this thesis.

Effect signatures define the type of interaction between effectful programs and their context. The following example defines an effect signature which enables effectful programs to emit integer values to their context.

```
effect Emit(x: Int): Unit
```

This signature defines the `Emit` effect with a single effect operation `Emit` which takes a parameter of type `Int` and returns a result of type `Unit`.

Effect operations are program statements that directly depend on or modify the evaluation context, making programs effectful. The following example generates a stream of numbers.

```
def generate(n: Int): Unit / Emit = {
  def loop(i: Int) =
    if(i < n) {
      do Emit(i); loop(i + 1)
    } else { () };
  loop(0)
}
```

The recursive function `loop` uses the effect operation `do Emit(i)` to emit the current value `i`. In effectful programs the order of operations matters, therefore statements are sequenced. The type signature of `generate` expresses that it takes a parameter of type `Int` and returns a result of type `Unit`. Moreover, it has the `Emit` effect whose meaning depends on the context.

Effect handlers provide meaning to effect operations. The following example handles the `Emit` effect by gathering all emitted values into a list.

```
def gather { prog: () ⇒ Unit / Emit }: List[Int] =
  try {
    prog(); Nil()
  } with Emit { (x) ⇒
    val xs = resume(()); Con(x, xs)
  }
```

The higher-order function `gather` takes a program `prog` which has the `Emit` effect. It calls `prog()` under a handler for the `Emit` effect and returns the empty list `Nil()`. The handler implementation of `Emit` takes the emitted value as a parameter `x`. It resumes the program with the unit value `resume(())` to get the rest of the list `xs` and constructs a list `Con(x, xs)`.

Putting these pieces together, the following example generates five values and gathers them into a list.

```
gather { generate(5) }
```

We call `gather` with a program that calls `generate(5)`. Here the effectful function `generate`, which has the `Emit` effect, and the handler function `gather`, which handles the `Emit` effect, meet. All effects are handled and the overall program is pure. It evaluates to the list `[0,1,2,3,4]`.

Effect handlers make effecful programs compositional. The following example transforms a stream of numbers.

```
def transform { func: Int ⇒ Int } { prog: () ⇒ Unit / Emit }:
  Unit / Emit =
    try {
      prog()
    } with Emit { (x) ⇒
      do Emit(func(x)); resume(())
    }
```

The function `transform` takes a function `func` and a program `prog`. The handler implementation takes the parameter `x`, applies the function `func` to it, emits the result, and resumes the computation. The function `transform` has the `Emit` effect, and also handles the `Emit` effect in the given program.

The following example generates five numbers, uses `transform` to square each of them, and gathers the results into a list.

```
gather { transform { (y) ⇒ y * y } { generate(5) } }
```

All effects are handled and the program evaluates to `[0,1,4,9,16]`. The meaning of the same `Emit` effect is different, depending on the context. The program `generate(5)` is used under two different handlers of the same `Emit` effect: an outer handler installed by `gather` and an inner handler installed by `transform`.

This running example demonstrates how it is possible to define and work with generators using effect handlers. Generators are usually built into programming languages, but with effect handlers they can be defined by users and shared in a library. The usefulness of effect handlers goes far beyond this simple example and generators. They subsume a number of useful language features [Pretnar, 2015, Dolan et al., 2015, Leijen, 2017a, Dolan et al., 2017, Kammar et al., 2013, Hillerström and Lindley, 2016, Piróg et al., 2018, Bračevac et al., 2018b]. Moreover, they enable less common programming patterns with non-trivial control flow. These more advanced control-flow abstractions are also user-definable and can be shared as libraries. It is naturally possible to combine these libraries and the corresponding domain-specific abstractions in one program.

Effect handlers are not merely an academic exercise. At the time of writing, they are starting to gain traction in industry as, indeed, effect handlers have many practically relevant applications.

1.2 Contributions

In this thesis we develop a compilation technique for lexical effect handlers. It rests on three key ideas: explicit capability passing, iterated continuation passing, and subregion evidence passing. The general theme is to make more and more information explicit throughout compilation. A compiler can then use this explicit information to optimize programs, sometimes completely eliminating all effect handlers and their associated runtime cost.

As we have seen, the meaning of effectful programs depends on their evaluation context. In languages with support for effect handlers, the handler implementations are part of this evaluation context. In prior work, language runtimes perform a dynamic lookup to find a matching handler implementation for an effect operation. These dynamic lookups incur a run-time penalty and they preclude compile-time optimizations.

To evaluate the call to an effect operation typically includes two tasks at runtime: firstly, performing a linear lookup through the evaluation context to find the corresponding effect handler and, secondly, capturing a segment of the context delimited by that very handler. In general, the full evaluation context can only be known at runtime. However, if certain information about the context is available at compile time, we can use it to specialize effectful programs.

1.2.1 Explicit Capability Passing

Chapter 2 presents our understanding of effects as capabilities. Effect handlers introduce term-level capabilities and effectful programs use these capabilities to perform effects. This first ingredient of our compilation technique makes the connection between effect operations and their corresponding handlers explicit.

The following example shows the result of transforming the above program to explicit capability-passing style.

$$\mathsf{gather}(\{\, \mathsf{emit}_1 \Rightarrow \mathsf{transform}(\mathsf{emit}_1,\ \{\, \mathsf{y} \Rightarrow \mathsf{y} * \mathsf{y} \,\},\ \{\, \mathsf{emit}_2 \Rightarrow \mathsf{generate}(\mathsf{emit}_2,\ 5) \,\}) \,\})$$

The function gather introduces a capability emit_1 which is passed to transform. The function transform introduces a capability emit_2 which is passed to generate. Explicit capability passing makes it possible to know which effect operation corresponds to which handler by lexical reasoning. This understanding helps programmers and compilers alike.

This chapter presents the following results:

- A formal presentation of the language Effekt which has lexical effect handlers and gurantees effect safety.
- An algorithmic effect system, which is rooted in our understanding of effects as capabilities.
- A calculus System Ξ in explicit capability-passing style, and a semantics that generates a fresh label for each handler instance at runtime.
- The type system of System Ξ does not include effects or effect types. Effect safety is established by treating capabilities as second class.
- A definition of the semantics of Effekt, a language with effect handlers, by translation into System Ξ, a language in explicit capability-passing style.
- A mechanized proof of soundness of System Ξ, a proof of well-typedness preservation of the translation, and a proof of effect safety of Effekt.

1.2.2 Iterated Continuation Passing

Chapter 3 develops a translation to iterated continuation-passing style in a typed setting from first principles. In iterated continuation-passing style, effectful programs receive not one but potentially multiple continuations. This second ingredient of our compilation technique makes complex control flow explicit.

The following example shows the function `gather` translated to explicit capability-passing style and to iterated continuation-passing style.

```
let gather prog = \k0 ⇒
  let emit x = \k1 ⇒ \k2 ⇒ k1 () (\xs ⇒ k2 (Con x xs))
  in prog emit (\u ⇒ \k ⇒ k Nil) k0
```

The effectful function `emit` takes a parameter `x` and two continuation `k1` and `k2`. The program `prog` receives the capability `emit` and two continuations. In continuation-passing style capturing the current continuation is immediate. The more general iterated continuation-passing style accounts for the nesting of handler.

This chapter presents the following results:

- A reconstruction of the well-known higher-order one-pass continuation-passing style translation which avoids generating administrative beta redexes.
- A generalization of this translation to iterated continuation-passing style and shows how to avoid administrative beta- and eta redexes in this more general setting.
- The concept of a stack shape, which is the list of answer types at delimiters, from innermost to outermost.
- Different parts of the program can take a different number of continuations, whereas previous work fixed the number of continuations for the whole program.
- The meta-language, as well as our object languages, are all typed and the translations preserve well-typedness by construction.

1.2.3 Subregion Evidence Passing

Chapter 4 draws a connection between effects and regions. Effect safety is ensured by type-level regions. Effect-polymorphic functions become region-polymorphic functions. This third ingredient of our compilation technique makes the nesting of handlers explicit.

The following example shows the function `generate` in explicit capability-passing style with explicit regions and explicit subregion evidence.

```
def generate[r, r₁; n₁ : r ⊑ r₁](emit : Emit[r₁], n : Int) : Unit at r {
  def loop[r₂; n₂ : r₂ ⊑ r₁](i : Int) at r₂ {
    if(i < n) {
      do emit[n₂](i); loop[r₂; n₂](i + 1)
    } else { return () }
  };
  loop[r; n₁](0)
}
```

It explicitly abstracts over regions (*i.e.*, r, r_1, and r_2) and subregion evidence (*i.e.*, n_1 and n_2). The capability `emit` can only be used in subregions of r_1. The local definition loop uses the capability emit. Therefore loop can only be used in subregions of r_1, which is expressed as the constraint $\mathsf{r}_2 \sqsubseteq \mathsf{r}_1$.

This chapter presents the following results:

- A formal presentation of Λ_{Cap}, a language with lexical effect handlers, first-class functions, and a type-and-effect system based on regions.
- An operational semantics for Λ_{Cap} as an abstract machine that generates a fresh label for each handler instance at runtime.
- Mechanized proofs of Progress and Preservation for this abstract machine. Effect safety follows as a simple corollary.
- A translation of Λ_{Cap} to pure System F in iterated continuation-passing style. The translation takes well-typed programs in Λ_{Cap} to well-typed terms in System F. This entails effect safety of Λ_{Cap} in yet another way.
- A proof that translated terms simulate terms under the operational semantics. This is surprising, since the operational semantics uses labels to find handlers on the stack, while the translation targets pure System F, without any labels, mutable state, or recursive types.
- Our compilation technique with explicit capability passing and iterated continuation passing correctly works in the presence of effect polymorphism and first-class functions.

1.2.4 Compiling Effect Handlers

Chapter 5 evaluates the compilation technique for lexical effect handlers presented in this thesis both theoretically and practically. It demonstrates that a compiler can exploit the information that we have made explicit to aggressively optimize programs at compile time. Indeed, under some conditions, we show that it is always possible to completely remove all handler abstractions. It also provides benchmark results, which are quite encouraging and have been independently reproduced by Karachalias et al. [2021].

The following example is the overall result of compiling the motivating program, which generates a list of squares, to simply-typed lambda calculus with the compilation technique presented in this thesis.

```
letrec loop i k1 k2 =
  if (i < 5)
  then loop (i + 1) k1 (\xs ⇒ k2 (Con (i * i) xs))
  else k1 () k2
in loop 0 (\u ⇒ \k ⇒ k Nil) (\z ⇒ z)
```

The program still evaluates to `[0,1,4,9,16]`. It consists of the recursive function `loop`. All capabilities, operations, and handlers are gone. The function `loop` receives two continuations `k1` and `k2` corresponding to the two handlers. The function `func` which squares its parameter has been inlined. No intermediate data structure is allocated.

This example demonstrates that stream fusion [Coutts et al., 2007, Kiselyov et al., 2017] emerges naturally from our compilation technique. Our compiliation technique is not specific to streams and does not treat streams specially. Other effect operations and their combinations fuse in a similar way.

This chapter presents the following results:

- The combination of explicit capability passing with iterated continuation passing and subregion evidence as a compilation technique for lexical effect handlers.
- A formal presentation of λ_{Cap} a language in explicit capability-passing style with a type-and-effect system where effects are stack shapes.
- A translation of λ_{Cap} to STLC, simply-typed lambda calculus.
- A refinement of λ_{Cap} to a subset $\lambda\!\!\lambda_{\mathsf{Cap}}$, for which full elimination of all handler abstractions is guaranteed.
- A translation of $\lambda\!\!\lambda_{\mathsf{Cap}}$ to $\mathsf{2STLC}$, a two-level lambda calculus.
- Both translations never fail, always terminate, and the generated programs are well-typed. Effect safety follows as a corollary.
- Benchmarks of the implementations of λ_{Cap} and $\lambda\!\!\lambda_{\mathsf{Cap}}$, which suggest that the code we generate is competitive with or faster than Koka, Multicore OCaml, and Chez Scheme using control effects.

2 Capability Passing

Abstract. In this chapter we introduce Effekt, a language with lexical effect handlers. We start with an informal introduction of programming with effect handlers in Effekt, before introducing it formally. We then formally introduce System Ξ, a language with lexical effect handlers in explicit capability-passing style. We define the semantics of Effekt as a translation to System Ξ. This translation makes the flow of capabilities explicit. We present a mechanized proof of soundness, which entails effect safety.

Explicit capability passing is the first ingredient of the compilation technique presented in this thesis.

Effect handlers need access to the current continuation. The semantics of System Ξ, presented in this chapter, is defined using multi-prompt delimited control. Each handler generates a fresh label at runtime. Effect operations use this label to capture the correct part of the runtime stack. In Chapter 3 we introduce iterated continuation-passing style, an alternative way of getting access to the current continuation.

To guarantee effect safety, in this chapter, we separate functions from values and treat all functions as second-class. While it is possible for functions to abstract over other functions, they can never be returned. In Chapter 4 we lift this restriction, and guarantee effect safety with a region system and explicit subregion evidence. Moreover, we show that the semantics with multi-prompt delimited control presented in this chapter is correctly simulated by a translation to iterated continuation-passing style.

In Chapter 5 we evaluate the compilation technique presented in this thesis. We start from a language in explicit capability-passing style as presented in this chapter. Being explicit about the flow of capabilities helps us to specialize programs to concrete handlers at compile time.

This chapter is based on the following publication: Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. “Effects as capabilities: Effect handlers and lightweight effect polymorphism”. *Proc. ACM Program. Lang.*, 4 (OOPSLA). DOI: https://doi.org/10.1145/3428194

2.1 Programming with Effect Handlers

In this section we introduce programming with effect handlers [Plotkin and Pretnar, 2013] in Effekt. In general, when programming with effect handlers, programs are structured into three components: *Effect signatures* that define available effect operations, *effectful programs* that use effect operations, and *effect handlers* that give meaning to effect operations.

As a running example, we adopt the example by Leijen [2016] and implement a parser combinator library using effect handlers. Our goal is to parse a list of numbers, while assembling the parser from individual reusable components.

2.1.1 The Fail Effect

Effect signatures provide the interface of effect operations, but not their implementation. In Effekt, effect signatures are declared as follows:

```
effect Fail[A](msg: String): A
```

The effect operation `Fail` aborts the current computation with a given message. It is polymorphic in its return type `A` so we can use it in any expression position. The following effectful function converts a string to an integer. It uses the `Fail` effect to signal that the conversion failed.

```
def stringToInt(str: String): Int / { Fail } = toInt(str) match {
  case Some(n) ⇒ n
  case None() ⇒ do Fail("cannot convert input to integer")
}
```

In case the value returned by the builtin function `toInt` is `None()`, we use the effect operation `Fail` to signal an error. We can freely compose programs that use effects. For instance, we can use the function `stringToInt` to convert and then add two numbers:

```
def perhapsAdd(): Int / { Fail } = stringToInt("1") + stringToInt("2")
```

The type of `perhapsAdd` communicates that it requires the calling context to handle the `Fail` effect, although `perhapsAdd` does not use the `Fail` effect directly. This requirement arises from the uses of `stringToInt`.

Handling effects is syntactically and conceptually similar to handling exceptions:

```
try { perhapsAdd() } with Fail { (msg) ⇒ 0 }
```

In case any conversion fails, this effect handler for the `Fail` effect returns `0` from the overall computation. The type of the program is `Int / {}`, it has no unhandled effects, and we can run it to get the result `3`.

2.1.2 The Read Effect

Effect handlers not only generalize exceptions, but can express many more effects. Another example is the `Read` effect that we will use to work with a pull-based stream of string values:

```
effect Read(): String
```

Using `Read` and `Fail`, we can express a parser that recognizes a number in an input stream:

```
def number() : Int / { Read, Fail } = stringToInt(do Read())
```

The return type of `number` communicates that we can only call it in a context that provides implementations for both `Read` and `Fail`.

We can handle the `Read` effect, for example, by always returning the string `"42"`:

```
def always42[R] { prog: () ⇒ R / { Read } }: R / {} =
  try { prog() } with Read { () ⇒ resume("42") }
```

This handler implementation illustrates the additional power of effect handlers over exception handlers: in an effect handler we can call `resume` to transfer control back to the call-site of the effect operation. While the handler for `Fail` did not use `resume`, in this example we resume the computation with `"42"`. Since `Read` returns a string, `resume` has type `String ⇒ R / {}`. The type signature of `always42` communicates that it will handle the `Read` effect of `prog`. The empty effect set in the return type signals that `always42` itself does not require any effects. Since it is polymorphic in the result type `R`, we know that it will call `prog` and handle the `Read` effect.

The block passed to `always42` can have additional effects that need to be handled at the call-site of `always42`. However, the implementation of `always42` cannot interfere with these additional effects.

```
try { always42 { number() } } with Fail { (msg) ⇒ 0 }
```

In this example, we handle the two effects `Read` and `Fail` that `number` requires with different handlers. Running it results in the integer `42`.

Effect handlers grant flexibility in the interpretation of effect operations. We can define another handler for the `Read` effect that reads from a given list.

```
def feed[R](input: List[String]) { prog: () ⇒ R / { Read } } = {
  var remaining = input;
  try { prog() } with Read { () ⇒ remaining match {
    case Nil() ⇒ do Fail("End of input")
    case Con(elem, rest) ⇒ remaining = rest; resume(elem)
  }}
}
```

This alternative handler for `Read` shows two interesting things: Firstly, it itself uses the `Fail` effect to signal an unexpected end of the input stream. The inferred type and effect are `R / { Fail }`. Secondly, it uses a mutable variable `remaining` to keep track of the position in the input stream. Mutable variables in Effekt are conceptually stack allocated, which guarantees a well-defined interaction with control effects.

2.1.3 The Fork Effect

The handler for `Fail` discarded the resumption, the handler for `Read` called it exactly once. This third example effect illustrates that it can be useful to call the resumption *more than once*. For this, we define the effect `Fork`, which returns a boolean value to model the outcome of a (potentially) non-deterministic choice:

```
effect Fork(): Bool
```

In Effekt, we can mix effect operations with other imperative language constructs like loops and references. For example, we can define a higher-order function `many` that calls a given program an unknown number of times, controlled by the `Fork` effect:

```
def many { prog: () ⇒ Unit / {} }: Unit / { Fork } =
  while (do Fork()) { prog() }
```

We use `many` to define a parser that reads arbitrarily many numbers and adds them. Note how using `many` feels as natural as using the built-in control operator `while`:

```
def numbers() = { var res = 0; many { res = res + number() }; res }
```

The inferred return type of `numbers` is `Int / { Fork, Fail, Read }`. One possible example handler performs a backtracking search to find the first success:

```
def backtrack[R] { prog: () ⇒ R / { Fail, Fork } }: Result[R] / {} =
  try { Success(prog()) }
  with Fail { (msg) ⇒ Failure(msg) }
  with Fork { () ⇒ resume(true) match {
    case Failure(msg) ⇒ resume(false)
    case Success(res) ⇒ Success(res)
  }}
```

The handler uses a data type that represents a potentially negative outcome.

```
type Result[R] { Success(res: R); Failure(msg: String) }
```

The handler `backtrack` handles *two* effects: `Fail` and `Fork`. At each choice, it first resumes with `true` and in case of failure resumes a second time with `false`. Any use of the `Fail` effect aborts the current search path with `Failure`. Different handlers for `Fork` and `Fail` would correspond to different search strategies.

2.1.4 Parsing

Parsers like `numbers` use the effects `Fail`, `Read`, and `Fork` that we group under an effect alias:

```
effect Parser = { Fail, Read, Fork }
```

To handle the `Parser` effect, we compose the handler implementations from this section:

```
def parse[R](input: List[String]) { prog: () ⇒ R / Parser } =
  backtrack { feed(input) { prog() } }
```

A parser handles `Read` by reading from the given list of strings. It handles all failures in `prog` and in `feed` using the implementation of backtracking search. By nesting `feed` inside of `backtrack`, the position in the input stream is automatically correctly backtracked when a choice is resumed a second time. The inferred return type and effect of `parse` is `Result[R] / {}`, the set of unhandled effects is empty. Running the program `parse(["1", "2"]) { numbers() }` results in `Success(3)`.

2.1.5 Section conclusion

Effect handlers generalize exception handlers and offer additional expressivity. This way, the advanced control flow in programs like `numbers` can be modularly described with user-defined combinators. Our backtracking implementation corresponds roughly to a hand-written recursive descent parser with the advantage that the decision for a parsing algorithm is not hard coded into parsers like `numbers`. The imperative parser combinators can easily be combined with other effects like mutable state, exceptions, or let-insertion. Abstractions like the `many` combinator can be shared in a library and reused across different domains.

2.2 The Language Effekt

In this section we formally introduce Effekt. In Section 2.3 we formally introduce System Ξ, a language where effectful programs are in explicit capability-passing style and in Section 2.4 we present a translation of Effekt to System Ξ. This translation makes the flow of capabilities explicit and is the first step of the compilation technique presented in this thesis.

2.2.1 Syntax

$$
\begin{array}{llcll}
\text{Statements} & s & ::= & \mathbf{val}\, x \,=\, s;\, s & \text{sequencing} \\
 & & \mid & e & \text{expressions} \\
 & & \mid & \mathbf{def}\, f\,(\overline{x : \tau},\, \overline{g : \sigma}) : \tau \,/\, \varepsilon \,=\, s;\, s & \text{block definition} \\
 & & \mid & f\,(\overline{e},\, \overline{g}) & \text{block call} \\
 & & \mid & \mathbf{effect}\, F\,(x : \tau) : \tau;\, s & \text{effect declaration} \\
 & & \mid & \mathbf{do}\, F\,(e) & \text{effect call} \\
 & & \mid & \mathbf{try}\, \{\, s \,\}\, \mathbf{with}\, F\, \{\, (x : \tau) \Rightarrow s \,\} & \text{effect handling} \\
\text{Expressions} & e & ::= & x \mid v & \\
\text{Values} & v & ::= & () \mid 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \ldots & \text{primitives}
\end{array}
$$

$$
\begin{array}{llcl}
\text{Value Types} & \tau & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \ldots \\
\text{Block Types} & \sigma & ::= & (\overline{\tau},\, \overline{\sigma}) \rightarrow \tau \,/\, \varepsilon \\
\text{Effect Sets} & \varepsilon & ::= & \{\, F_1, \ldots, F_n \,\}
\end{array}
$$

$$
\begin{array}{llcl}
\text{Value Environment} & \Gamma & ::= & \emptyset \mid \Gamma,\, x : \tau \\
\text{Block Environment} & \Delta & ::= & \emptyset \mid \Delta,\, f : \sigma \\
\text{Effect Environment} & \Sigma & ::= & \emptyset \mid \Sigma,\, F : \tau \rightarrow \tau
\end{array}
$$

$$
\begin{array}{ll}
\text{Expression Variables} & x,\, y \in \mathsf{x},\, \mathsf{y} \\
\text{Block Variables} & f,\, g \in \mathsf{f},\, \mathsf{g} \\
\text{Operations} & F \in \mathsf{Fail},\, \mathsf{Fork},\, \ldots
\end{array}
$$

Figure 2.1: Syntax of the source language Effekt.

Figure 2.1 defines the syntax of Effekt. Like other languages with effect handlers [Hillerström et al., 2017] it is presented in fine-grain call-by-value [Levy et al., 2003]. That is, we syntactically distinguish between statements, which can have control effects, and expressions, which can not. In this sense, our statements are “serious” while expressions are “trivial” [Reynolds, 1972].

Statements

We sequence two statements with **val** $x = s_0;\ s_1$, where the result of s_0 will be bound to the variable x in s_1. The syntactic form **def** $f(\overline{x : \tau},\ \overline{g : \sigma}) : \tau \,/\, \varepsilon = s_0;\ s$ defines a block f, binding a fixed, but arbitrary number of value variables x as well as block variables g. Calling blocks is denoted $f(\overline{e},\ \overline{g})$, providing potentially multiple value arguments as expressions e and block arguments as block variables g. Without loss of generality, we do not allow passing *anonymous* blocks and require that all blocks are named, before passing them to a call. This convention significantly simplifies the presentation of the typing rules and the translation. Effect declarations are statements of the form **effect** $F(x : \tau) : \tau;\ s$, which means that effects can be declared locally. Effect calls (*i.e.*, **do** $F(e)$) only take a single expression argument. While the restriction to one argument is insignificant, it is important that effect operations only take expressions as arguments and never blocks. Otherwise blocks could escape their scope through the effect operation [Brachthäuser et al., 2020], violating effect safety. Finally, the statement **try** $\{\ s_0\ \}$ **with** $F\ \{\ (x : \tau) \Rightarrow s\ \}$ expresses that effect calls to F in the handled statement s_0 will be handled by the handler $\{\ (x : \tau) \Rightarrow s\ \}$. The special block variable resume is available in the handler implementation s.

Expressions

Effect safety of Effekt rests on the property that all functions (blocks) are second class. Consequently, blocks are syntactically *neither* values *nor* expressions. Only primitive constants are values. Similarly, we distinguish syntactically between variables that stand for values (x, y, ...) and variables that stand for blocks (f, g, ...). As usual, we follow Barendregt [1992] and require that all expression variables, block variables, and operation names are globally unique.

Types

The meta variable τ describes *value types*, which we use to type expressions. The meta variable σ describes *block types*, which we use to type blocks. Expressions are first-class, while blocks are second-class [Osvald et al., 2016]. A block type (*i.e.*, $(\overline{\tau},\ \overline{\sigma}) \to \tau \,/\, \varepsilon$) takes expressions of types $\overline{\tau}$ and blocks of types $\overline{\sigma}$ as parameters. The return type τ of a block indicates that only values (and not blocks) can be returned. The block type also mentions the effects ε that need to be handled by the caller. Effects $\varepsilon = \{\ F_1,\ \ldots,\ F_n\ \}$ are (closed) sets of operation names F_i. This is different from languages that base their effect systems on row polymorphic records where effect operations can occur multiple times [Leijen, 2017b] or effects are annotated with presence/absence information [Hillerström and Lindley, 2016]. Modeling effects as sets greatly simplifies typing as no special unification rules are needed [Leijen, 2005].

2.2.2 Typing

Figure 2.2 defines the typing rules of Effekt. Effekt features lexical effect handlers. Consequently, effect types express which capabilities a computation requires from its

Statement Typing

$$\boxed{\begin{array}{cccccccc} \Gamma \mid & \Delta \mid & \Sigma & \vdash & s & : & \tau \mid & \varepsilon \\ \uparrow & \uparrow & \uparrow & & \uparrow & & \downarrow & \downarrow \end{array}}$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash s_0 : \tau_0 \mid \varepsilon_0 \qquad \Gamma, x : \tau_0 \mid \Delta \mid \Sigma \vdash s_1 : \tau_1 \mid \varepsilon_1}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{val}\, x = s_0;\, s_1 : \tau_1 \mid \varepsilon_0 \cup \varepsilon_1}\ [\textsc{Val}]$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \mid \Delta \mid \Sigma \vdash e : \tau \mid \emptyset}\ [\textsc{Expr}]$$

$$\frac{\Gamma, \overline{x : \tau} \mid \Delta, \overline{g : \sigma} \mid \Sigma \vdash s_0 : \tau_0 \mid \varepsilon_0' \qquad \Gamma \mid \Delta, f : (\overline{\tau}, \overline{\sigma}) \to \tau_0 \,/\, \varepsilon_0 \mid \Sigma \vdash s : \tau \mid \varepsilon}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{def}\, f(\overline{x : \tau}, \overline{g : \sigma}) : \tau_0 \,/\, \varepsilon_0 = s_0;\, s : \tau \mid (\varepsilon_0' \setminus \varepsilon_0) \cup \varepsilon}\ [\textsc{Def}]$$

$$\frac{\overline{\Gamma \vdash e : \tau} \qquad \overline{\Delta(g) = \sigma} \qquad \Delta(f) = (\overline{\tau}, \overline{\sigma}) \to \tau \,/\, \varepsilon}{\Gamma \mid \Delta \mid \Sigma \vdash f(\overline{e}, \overline{g}) : \tau \mid \varepsilon}\ [\textsc{BlockCall}]$$

$$\frac{\Gamma \mid \Delta \mid \Sigma, F : \tau_1 \to \tau_0 \vdash s_2 : \tau_2 \mid \varepsilon_2 \qquad F \notin \mathsf{ftv}(\varepsilon_2)}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{effect}\, F(x_1 : \tau_1) : \tau_0;\, s_2 : \tau_2 \mid \varepsilon_2}\ [\textsc{Effect}]$$

$$\frac{\Sigma(F) = \tau_1 \to \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{do}\, F(e_1) : \tau_0 \mid \{ F \}}\ [\textsc{EffectCall}]$$

$$\frac{\begin{array}{c} \Sigma(F) = \tau_1 \to \tau_0 \qquad \Gamma \mid \Delta \mid \Sigma \vdash s_0 : \tau \mid \varepsilon_0 \\ \Gamma, x_1 : \tau_1 \mid \Delta, \mathit{resume} : (\tau_0) \to \tau \,/\, \emptyset \mid \Sigma \vdash s : \tau \mid \varepsilon \end{array}}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{try}\, \{ s_0 \}\, \textbf{with}\, F\, \{ (x_1 : \tau_1) \Rightarrow s \} : \tau \mid (\varepsilon_0 \setminus \{ F \}) \cup \varepsilon}\ [\textsc{Try}]$$

Expression Typing

$$\boxed{\begin{array}{cccc} \Gamma & \vdash & e & : \tau \\ \uparrow & & \uparrow & \downarrow \end{array}}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}}\ [\textsc{Lit}] \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ [\textsc{Var}]$$

Figure 2.2: Typing rules of the source language Effekt.

context. This intuition will be useful when we discuss the details of the typing rules. There are two judgments, one for expressions and one for statements.

Expression Typing

The judgment for expressions $\Gamma \vdash e : \tau$ assigns a value type τ to an expression e in value environment Γ. Typing of expressions only requires a value environment, since expressions cannot mention any blocks or effects. Furthermore, since expressions do not have control effects, expression typing computes a value type τ without any effects. The typing rules for expressions are completely standard.

Statement Typing

The judgment for statements $\Gamma \mid \Delta \mid \Sigma \vdash s : \tau \mid \varepsilon$ computes a value type τ and a set of required capabilities ε for the statement s. It uses three environments, a value environment Γ, a block environment Δ, and an effect environment Σ.

Rule Val types sequencing of statements. It accumulates all required capabilities of the statements s_0 and s_1 by taking the union of the corresponding effect sets ε_0 and ε_1. Rule Expr types an expression statement by assigning the empty set of effects.

Rule Def types block definitions. It is a bit more involved and requires some explanation. The capabilities a block requires can be provided in two different ways. Firstly, the block can mention a required effect in its type, which means that the effect is handled *dynamically* and the capability needs to be provided by the caller. Secondly, all capabilities that are not part of the annotated type need to be provided by the context at the *definition site* of the block. This can be seen in analogy to term-level variables: free variables in a function body can either be bound as parameters of the function, or they are bound in the context of the definition site. Because of this analogy, this form of effect handlers are called *lexical*.

Programmers can control which effects are handled at the call-site (*i.e.*, ε_0), and which effects are *free* (*i.e.*, $\varepsilon_0' \setminus \varepsilon_0$) and need to be handled at the definition site of a block. Operationally, as we will see in Section 2.4, the block will close over the free effects.

Since all complications are part of rule Def, typing block calls (rule BlockCall) takes a familiar form, simply checking whether the argument types conform to the annotated parameter types. The decision whether effects of block arguments are handled dynamically or lexically is clear from the types at their definitions.

Example 1. The following example illustrates how the Def rule works with our desugaring of anonymous blocks:

```
def optionally { prog: () → Int / { Fail } }: Option[Int] / {} = ...

optionally { () ⇒ if (do Fork()) do Fail("failed") else 42 }
```

We desugar this example to bind the block to a fresh name `anon` with exactly the type of the block parameter of `optionally`.

```
def optionally { prog: () → Int / { Fail } }: Option[Int] / {} = ...

def anon(): Int / { Fail } = if (do Fork()) do Fail("failed") else 42;
optionally(anon)
```

The example program has the overall type `Option[Int] / { Fork }`. The required capability Fail is provided by function `optionally`, while Fork is free and has to be provided by the context of the definition of `anon`.

The last three rules are concerned with effect declaration, use, and handling. Rule Effect extends the effect environment Σ, bringing the effect operation F into scope. The side-condition $F \notin @ftv(\varepsilon_2)$ corresponds to the standard check that type variables should not leave the scope in which they are defined [Eisenberg et al., 2018]. Symmetrically, rule EffectCall requires the effect signature to be lexically in scope when an effect operation is used. Interestingly, in our capability-oriented formalization, this simple check suffices to express locally defined effects, where other languages require sophisticated use of existential quantification [Biernacki et al., 2019a].

Lastly, rule Try types handling of effects. The handled statement s_0 is assigned return type τ and effects ε_0. In Effekt, after typing s_0, the handled effect is subtracted from the resulting set of effects ε. This, again, is an important difference compared to languages based on row polymorphism [Leijen, 2017b] where, by unification, the effect type of s_0 would necessarily include F. The body of the handler s can assume that the variable x_1 has type τ_1 and that the block variable *resume* has type $(\tau_0) \rightarrow \tau \,/\, \emptyset$. The latter might come with surprise: one might expect that the continuation still has effects. However, we can see that all effects in *resume* must be handled outside the corresponding **try** statement. In particular, the body of the operation clause does not have to (and even cannot) handle any effects in *resume*. Typing the continuation with the empty set of effects is safe since it is a block and cannot leave the scope of its definition. Finally, the resulting set of effects is the set of effects ε_0 of the handled statement, without F, but including all effects ε used by the effect operation.

2.3 The Core Language System Ξ

To specify the semantics of Effekt, we translate it to a core language: System Ξ. This section presents its syntax and type system, sketches its operational semantics, and states semantic soundness. Section 2.4 then defines the translation of Effekt to System Ξ and shows it preserves well-typedness. Effect safety of Effekt follows as a corollary. We provide a detailed discussion of the operational semantics of a more general language with lexical effect handlers in Section 4.2.

We have mechanized the type system and operational semantics of System Ξ in the dependently typed programming language Idris [Brady, 2013].

Statements	s	$::=$	$\textbf{val}\, x \;=\; s;\, s$	sequencing
		$\mid$	$\textbf{return}\, e$	returning
		$\mid$	$\textbf{def}\, f \;=\; b;\, s$	block definition
		$\mid$	$b(\overline{e},\, \overline{b})$	block call
		$\mid$	$\textbf{handle}\,\{\, F \Rightarrow s \,\}\,\textbf{with}\,\{\, (x,\, k) \Rightarrow s \,\}$	handler
Expressions	e	$::=$	$x \mid v$	
Expression Values	v	$::=$	$() \mid 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \ldots$	constants
Blocks	b	$::=$	$f \mid w$	
Block Values	w	$::=$	$\{\, (\overline{x : \tau},\, \overline{f : \sigma}) \Rightarrow s \,\}$	

Value Types	τ	$::=$	$\mathsf{Int} \mid \mathsf{Bool} \mid \ldots$	Value Environment	Γ	$::=$	$\emptyset \mid \Gamma,\, x : \tau$
Block Types	σ	$::=$	$(\overline{\tau},\, \overline{\sigma}) \to \tau$	Block Environment	Δ	$::=$	$\emptyset \mid \Delta,\, f : \sigma$

Expression Variables $x,\, y \in \mathsf{x},\, \mathsf{y}$ Block Variables $f,\, g,\, k,\, F \in \mathsf{f},\, \mathsf{g},\, \mathsf{k},\, \mathsf{Fail},\, \mathsf{Fork},\, \ldots$

Figure 2.3: Syntax of System Ξ.

2.3.1 Syntax

Figure 2.3 defines the syntax of System Ξ. Like Effekt, the core language is in fine-grain call-by-value. Also like Effekt, it distinguishes between expressions e and blocks b. Unlike Effekt, however, the core language supports effect handlers in *explicit capability-passing style*. That is, effect operations are represented by blocks, which are introduced by the corresponding handler and passed as additional arguments. As a consequence, System Ξ does not distinguish between named blocks (that is, function definitions), anonymous blocks, and effect operations – all three are represented by the syntactic category of blocks b.

Blocks b can either be block values w of the form $\{\, (\overline{x : \tau},\, \overline{f : \sigma}) \Rightarrow s \,\}$ or block variables f. Note how block variables (*e.g.*, Fork or Fail) in System Ξ may have the names of effect operations in Effekt. Local blocks are defined with $\textbf{def}\, f \;=\; b;\, s$, binding block b to the name f in scope of the statement s. Block calls in System Ξ, of the form $b(\overline{e},\, \overline{b})$, subsume block and effect calls of Effekt. Arguments can be an arbitrary number of value arguments and block arguments. Finally, the handle statement $\textbf{handle}\,\{\, F \Rightarrow s_0 \,\}\,\textbf{with}\,\{\, (x,\, k) \Rightarrow s \,\}$ binds the block variable F in the handled statement s_0. The value parameter x and the continuation block k are bound in the handler implementation s.

2.3.2 Typing

The typing rules of System Ξ are defined in Figure 2.4. Like the source language, System Ξ distinguishes between two kinds of types: value types τ and block types σ. Importantly, block types now do not mention any effects, but only map value- and

Statement Typing

$$\boxed{\Gamma \mid \Delta \vdash s : \tau}$$

$$\frac{\Gamma \mid \Delta \vdash s_0 : \tau_0 \quad \Gamma, x : \tau_0 \mid \Delta \vdash s_1 : \tau_1}{\Gamma \mid \Delta \vdash \textsf{val}\, x = s_0;\, s_1 : \tau_1}\,[\textsc{Val}]$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \mid \Delta \vdash \textsf{return}\, e : \tau}\,[\textsc{Ret}] \qquad \frac{\Gamma \mid \Delta \vdash b : \sigma \quad \Gamma \mid \Delta, f : \sigma \vdash s : \tau}{\Gamma \mid \Delta \vdash \textsf{def}\, f = b;\, s : \tau}\,[\textsc{Def}]$$

$$\frac{\Gamma \mid \Delta \vdash b : (\overline{\tau}, \overline{\sigma}) \to \tau_0 \quad \overline{\Gamma \vdash e : \tau} \quad \overline{\Gamma \mid \Delta \vdash b : \sigma}}{\Gamma \mid \Delta \vdash b(\overline{e}, \overline{b}) : \tau_0}\,[\textsc{Call}]$$

$$\frac{\Gamma \mid \Delta, F : \tau_1 \to \tau_0 \vdash s_0 : \tau \qquad \Gamma, x : \tau_1 \mid \Delta, k : \tau_0 \to \tau \vdash s : \tau}{\Gamma \mid \Delta \vdash \textsf{handle}\,\{\, F \Rightarrow s_0 \,\}\,\textsf{with}\,\{\, (x, k) \Rightarrow s \,\} : \tau}\,[\textsc{Handle}]$$

Block Typing

$$\boxed{\Gamma \mid \Delta \vdash b : \sigma}$$

$$\frac{\Delta(f) = \sigma}{\Gamma \mid \Delta \vdash f : \sigma}\,[\textsc{BlockVar}] \qquad \frac{\Gamma, \overline{x : \tau} \mid \Delta, \overline{f : \sigma} \vdash s_0 : \tau_0}{\Gamma \mid \Delta \vdash \{\, (\overline{x : \tau}, \overline{f : \sigma}) \Rightarrow s_0 \,\} : (\overline{\tau}, \overline{\sigma}) \to \tau_0}\,[\textsc{Block}]$$

Expression Typing

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,[\textsc{Var}] \qquad \frac{}{\Gamma \vdash n : \textsf{Int}}\,[\textsc{Lit}]$$

Figure 2.4: Type system of System Ξ.

block parameter types to a resulting value type τ. Furthermore, inspecting the type system of System Ξ, we can see that it does not include an effect system. Effect safety is established by treating blocks as second class.

The type system of System Ξ has three judgments, one for each syntactic category. Statements and blocks are typed against two environments: environment Γ for value variables and environment Δ for block variables. Since effects are translated to blocks, the signature environment Σ is not required anymore.

Besides distinguishing between values and blocks and using separate environments, the typing rules for sequencing (Val), returning (Ret), block definitions (Def), and block calls (Call) are completely standard. They correspond to the rules of Effekt but without any tracking of effects.

Rule Handle is central to the calculus. We handle statement s_0 with the handler implementation s. The handler introduces a capability and binds it to the operation name F, which is brought into scope as a block variable in the handled statement s_0. In the handler implementation s, the parameter of the effect operation x has type τ_1 and the continuation k is an ordinary block variable of type $\tau_0 \to \tau$. The (answer) type τ appears four times in this rule: As the return type of the overall statement, the return type of the handled statement, the result type of the continuation, and the return type of the handler implementation – all have to agree.

2.3.3 Operational Semantics

We give the semantics of System Ξ as a small-step operational semantics using evaluation contexts [Wright and Felleisen, 1994]. To allow capturing and resuming continuations, the semantics of System Ξ follows the generative semantics presented by Biernacki et al. [2019b], who in turn present a variant of multi-prompt delimited control [Gunter et al., 1995]. Like previous presentations of effect handlers [Kammar et al., 2013], capturing a continuation removes the corresponding delimiter and resuming reinstalls the delimiter. This corresponds to a multi-prompt variant of the control operator $\textbf{shift}_0$ [Danvy and Filinski, 1989]. Our presentation requires two additional runtime constructs that only appear during evaluation: delimiters and capabilities.

Labels Both runtime constructs refer to unique runtime labels l, generated during reduction. We only require that labels can be compared for equality and that we are able to generate fresh labels at runtime. We represent concrete labels as hexadecimal number (*e.g.*, `@a5f`) to highlight that they are created at runtime.

Delimiters The additional statement $\#_l \{ s \}$ represents a *delimiter* that delimits a statement s at a given label l (or *prompt* in the terminology of Felleisen [1988], Sitaram [1993], and Gunter et al. [1995]).

Capabilities The additional block value $\textbf{cap}_l \{ (x, k) \Rightarrow s \}$ represents a *capability*, which is a pair of a label l and a handler implementation s [Brachthäuser and Schuster, 2017]. Calling a capability captures the stack segment up to the next dynamically enclosing delimiter for the label l, reifies it as a continuation, and binds it to k.

Reduction Rules

This presentation of the operational semantics follows Gunter et al. [1995] and is based on *delimited evaluation contexts* H_l where the label l does not appear in any delimiters in H_l. It is used to guarantee that captured continuations are always delimited by the dynamically closest delimiter for a label.

Handling introduces delimiters Rule *(handle)* creates a fresh runtime label l, delimits the handled statement s_0 with this label, and substitutes a capability that refers to l for the block variable F.

$$\textit{(handle)} \qquad \begin{array}{l} \textbf{handle}\,\{\, F \Rightarrow s_0 \,\}\,\textbf{with}\,\{\,(x,\ k) \Rightarrow s\,\} \ \longrightarrow \\ \#_l\,\{\, s_0[F \mapsto \textbf{cap}_l\,\{\,(x,\ k) \Rightarrow s\,\}]\,\} \qquad l\ \textit{fresh} \end{array}$$

Since the label is fresh, the capability is only valid in the dynamic region delimited by $\#_l$. Calling the capability outside of the region will lead to a stuck term. Our semantics is *generative*: reducing the same **handle** statement twice will introduce two distinct runtime labels [Biernacki et al., 2019b].

Capabilities capture the continuation The most interesting rule *(capture)* captures part of the context:

$$\textit{(capture)} \qquad \begin{array}{l} \#_l\,\{\,\mathsf{H}_l[\ (\textbf{cap}_l\,\{\,(x,\ k) \Rightarrow s\,\})(v)\ ]\,\} \ \longrightarrow \\ s[x \mapsto v,\ k \mapsto \{\, y \Rightarrow \#_l\,\{\,\mathsf{H}_l[\ \textbf{return}\ y\ ]\,\}\ \}] \end{array}$$

The application of a capability to a value (*e.g.*, $(\textbf{cap}_l\,\{\,(x,\ k) \Rightarrow s\})(v)$) itself is *not* a redex. This highlights the essence of effects: they depend on (and modify) the context they are evaluated in. The application of a capability with label l is only meaningful in a context, which is delimited at label l. This becomes visible in rule *(capture)*, where the delimiter $\#_l$, the delimited context H_l, and the capability application together form a redex. We reify this context as a continuation and substitute it (as well as the argument v) in the body of the handler implementation. Effect safety means that applications of a capability with label l only occur in a context with a delimiter at l (Theorem 4).

Only values can leave delimiters Once a statement is reduced to a value, delimiters are discarded:

$$\textit{(ret)} \qquad \#_l\,\{\,\textbf{return}\ v\,\} \ \longrightarrow\ \textbf{return}\ v$$

Since blocks and capabilities are no expression values, they cannot be returned.

Example 2. The following example illustrates the operational semantics of capturing continuations. We assume the effect operation Yield has type $\mathsf{Int} \to \mathsf{Int}$.

$$\textbf{handle}\,\{\,\mathsf{Yield} \Rightarrow \textbf{val}\,\mathsf{x} \;=\; \mathsf{Yield}(20);\ \textbf{return}\,\mathsf{x} * 2\,\}\,\textbf{with}\,\{\,(\mathsf{x},\,\mathsf{k}) \Rightarrow \mathsf{k}(\mathsf{x} + 1)\,\}$$

Reducing **handle** introduces a fresh label (*e.g.*, `@a1`) and uses it to delimit the handled program. It also introduces a capability and substitutes it for Yield:

$$\#_{\texttt{@a1}}\,\{\,\textbf{val}\,\mathsf{x} \;=\; (\textbf{cap}_{\texttt{@a1}}\,\{\,(\mathsf{x},\,\mathsf{k}) \Rightarrow \mathsf{k}(\mathsf{x} + 1)\,\})(20);\ \textbf{return}\,\mathsf{x} * 2\,\}$$

Applying rule *(capture)*, we obtain (captured stack segment highlighted in *gray*):

$$\mathsf{k}(20 + 1) \quad \text{where} \quad \mathsf{k} \;=\; \{\,\mathsf{y} \Rightarrow \#_{\texttt{@a1}}\,\{\,\textbf{val}\,\mathsf{x} \;=\; \textbf{return}\,\mathsf{y};\ \textbf{return}\,\mathsf{x} * 2\,\}\,\}$$

Further reducing the application results in

$$\#_{\texttt{@a1}}\,\{\,\textbf{val}\,\mathsf{x} \;=\; \textbf{return}\,21;\ \textbf{return}\,\mathsf{x} * 2\,\}$$

where we proceed to reduce under the delimiter to obtain $\#_{\texttt{@a1}}\,\{\,\textbf{return}\,42\,\}$, and finally remove the delimiter to get the result 42. As can be seen from the example, capturing the continuation removes the corresponding delimiter $\#_{\texttt{@a1}}$ and calling the continuation reinstalls it. This treatment of delimiters together with capability passing models *deep handlers* [Kammar et al., 2013].

2.3.4 Soundness

In our mechanized formalization, we represent System Ξ terms by their typing derivations [Benton et al., 2012] and show progress constructively by implementing the semantics as a total step function. One important class of stuck terms are capability applications without a corresponding delimiter.

Definition 3 (Undelimited Label)**.** A statement s contains an undelimited label l, if it has the form $\mathsf{H}_l[(\textbf{cap}_l\,\{\,(x,\,k) \Rightarrow s'\,\})(v)]$.

In our operational semantics, reducing a redex never produces a newly undelimited label. In the type system for System Ξ extended with runtime constructs, we add an additional *label context* Ξ to the typing judgement, which now has the form $\Gamma \mid \Delta \mid \Xi \;\vdash\; s \;:\; \tau$. All typing rules in Figure 2.4 ignore Ξ and simply pass it to the premises. We use this label context in our proof to formally capture the invariant that there are no undelimited labels.

Starting from an empty label context, closed and well-typed System Ξ programs either are values or we can take a step. Here the relation $\longmapsto$ describes congruence, that is, reduction under a context.

Theorem 4 (Progress of System Ξ)**.**
If $\emptyset \mid \emptyset \mid \emptyset \vdash\; s \;:\; \tau$, then s is of the form ***return*** *v or $s \longmapsto s'$.*

Our mechanized formalization establishes preservation by indexing System Ξ programs with their type. Performing a reduction step on a statement preserves its type:

$$
\begin{array}{lcl}
\mathcal{T}[\![\,(\overline{\tau},\,\overline{\sigma}) \rightarrow \tau_0 \,/\, \{\, F_1,\, \ldots .\, F_n \,\}\,]\!] & = & (\overline{\tau},\, \overline{\mathcal{T}[\![\sigma]\!]},\, \mathcal{T}[\![F_1]\!],\, \ldots,\, \mathcal{T}[\![F_n]\!]) \rightarrow \tau_0 \\
\mathcal{T}[\![\, F \,]\!] & = & (\tau_1) \rightarrow \tau_0 \\
\quad \text{where} \quad \Sigma(F) \;=\; \tau_1 \rightarrow \tau_0 & & \\
\mathcal{T}[\![\, \{F_1,\, \ldots,\, F_n\} \,]\!] & = & F_1 \;:\; \mathcal{T}[\![F_1]\!],\, \ldots,\, F_n \;:\; \mathcal{T}[\![F_n]\!] \\
\mathcal{S}[\![\, \textbf{val}\, x \;=\; s_0;\; s_1 \,]\!] & = & \textbf{val}\, x \;=\; \mathcal{S}[\![\, s_0 \,]\!];\; \mathcal{S}[\![\, s_1 \,]\!] \\
\mathcal{S}[\![\, e \,]\!] & = & \textbf{return}\; e \\
\mathcal{S}[\![\, \textbf{def}\, f(\overline{x},\, \overline{g}) \;:\; \tau_0 \,/\, \varepsilon_0 \;=\; s_0;\; s \,]\!] & = & \textbf{def}\, f \;=\; \{\, (\overline{x},\, \overline{g},\, F_1,\, \ldots,\, F_n) \Rightarrow \mathcal{S}[\![\, s_0 \,]\!] \,\};\; \mathcal{S}[\![\, s \,]\!] \\
\quad \text{where} \quad \varepsilon_0 \;=\; \{\, F_1,\, \ldots .\, F_n \,\} & & \\
\mathcal{S}[\![\, f(\overline{e},\, \overline{g}) \,]\!] & = & f(\overline{e},\, \overline{g},\, F_1,\, \ldots F_n) \\
\quad \text{where} \quad f \;:\; (\overline{\tau},\, \overline{\sigma}) \rightarrow \tau_0 \,/\, \{\, F_1,\, \ldots F_n \,\} & & \\
\mathcal{S}[\![\, \textbf{effect}\, F(x_1 \;:\; \tau_1) \;:\; \tau_0;\; s \,]\!] & = & \mathcal{S}[\![\, s \,]\!] \\
\mathcal{S}[\![\, \textbf{do}\, F(e_1) \,]\!] & = & F(e_1) \\
\mathcal{S}[\![\, \textbf{try}\, \{\, s_0 \,\}\, \textbf{with}\, \{\, F(x) \Rightarrow s \,\} \,]\!] & = & \textbf{handle}\, \{\, F \Rightarrow \mathcal{S}[\![\, s_0 \,]\!] \,\}\, \textbf{with}\, \{\, (x,\, \mathit{resume}) \Rightarrow \mathcal{S}[\![\, s \,]\!] \}
\end{array}
$$

Figure 2.5: Translation of Effekt to System Ξ.

Theorem 5 (Preservation of System Ξ).
If $\emptyset \mid \emptyset \mid \emptyset \vdash \; s \;:\; \tau$ *and* $s \longmapsto s'$ *then* $\emptyset \mid \emptyset \mid \emptyset \vdash \; s' \;:\; \tau$.

In particular, reduction also preserves the (empty) label context. That is, from progress and preservation follows effect safety: programs are never stuck on an undelimited label.

2.4 Translation of Effekt to System Ξ

Having introduced both Effekt and System Ξ formally, we now show how to make the flow of capabilities explicit by translating Effekt into System Ξ, *i.e.* into explicit capability-passing style. The translation is type directed and operates on typing derivations. Intuitively, where in Effekt the effect types indicate that a computation requires capabilities to be available in its context, in System Ξ we explicitly pass such capabilities as additional arguments. Figure 2.5 defines the translation. The translation is defined on types and on statements. We neither translate expressions, nor their types, since the language of expressions is the same in Effekt and System Ξ.

Translation of types We translate blocks that require a set of effects $\{\, F_1,\, \ldots,\, F_n \,\}$ to blocks that receive n additional block arguments – one for each member of the set. The translation of effect sets to additional arguments can be seen in the translation of block types, of block definitions, and of block calls. Names of operations in Effekt are now names of block variables in System Ξ. For the translation, we assume a canonical ordering of effects in each ε. In our implementation, it suffices to choose an arbitrary but fixed ordering for each type signature. The translation of types extends to environments and sets of effects. In particular, we translate the sets of effects ε of Effekt into block environments Δ of System Ξ translating each effect operation to a binding in the block environment. We assume that names of blocks and names of effect operations are disjoint and no name conflicts arise.

Translation of terms The translation of block definitions uses type information to add additional capability parameters to the block f. Symmetrically, the translation of application adds additional arguments. Assuming $\Sigma(\mathsf{Fail}) = (\mathsf{String}) \to \mathsf{Int}$, the program of Example 1 translates to:

$$\textbf{def}\ \mathsf{optionally}\ =\ \{\,(\mathsf{prog} :\ ((\mathsf{String}) \to \mathsf{Int}) \to \mathsf{Int}) \Rightarrow \ldots\,\}$$

$$\textbf{def}\ \mathsf{anon}\ =\ \{\,(\mathsf{Fail} :\ (\mathsf{String}) \to \mathsf{Int}) \Rightarrow \textbf{if}\ (\mathsf{Fork}())\ \textbf{then}\ \mathsf{Fail}(\text{"failed"})\ \textbf{else}\ 42\,\}$$
$$\mathsf{optionally}(\mathsf{anon})$$

Effect calls translate to ordinary block calls, where the name of the called block is the same as the name of the effect operation (*e.g.*, Fork or Fail). The effect system of Effekt and the translation guarantee that a block with the name of the effect operation is in scope. Effect types and their declarations disappear during translation. Translating the **try** statement of Effekt into the **handle** statement of System Ξ makes two things explicit: Firstly, the **handle** statement now explicitly binds the capability F as a block in the handled statement s_0. Secondly, the continuation *resume* is bound explicitly as a block variable in the body s.

2.4.1 Well-Typedness Preservation

The translation of Effekt to System Ξ in explicit capability-passing style preserves well-typedness:

Theorem 6 (Translation preserves well-typedness)**.**
If $\Gamma \mid \Delta \mid \Sigma \ \vdash\ s : \tau \mid \varepsilon$ *, then* $\Gamma \mid \mathcal{T}[\![\Delta]\!], \mathcal{T}[\![\varepsilon]\!] \mid \emptyset \ \vdash\ \mathcal{S}[\![s]\!] : \tau$*.*

Proof. Straightforward induction over the typing derivations. □

The translated program $\mathcal{S}[\![\, s \,]\!]$ is valid under the empty label context $\Xi = \emptyset$, *i.e.* does not contain any undelimited labels. This is obvious as the translation (Figure 2.5) never introduces any labels, delimiters, or capabilities. Those are only introduced at runtime by reducing **handle** statements.

2.4.2 Semantic Soundness of Effekt

We define the semantics of Effekt as the composition of the translation to System Ξ and the semantics of System Ξ. This presentation emphasizes our capability-based understanding of effects by translating them to explicitly passed blocks. Semantic soundness of Effekt directly follows from preservation of well-typedness (Theorem 6) and soundness of System Ξ. Effect safety follows immediately. We can identify two potential sources of runtime errors that would violate effect safety:

Unhandled effects Effects in Effekt might be unhandled, that is there is no enclosing effect handler that would handle the effect. In our translation, effect operations are translated to block calls. Unhandled effects thus would correspond to unbound block variables. The type-system of System Ξ and well-typedness preservation guarantees that such programs cannot be expressed.

Escaping capabilities Effects are translated to capabilities and those contain labels. Those capabilities could leave the region of the corresponding delimiter, leading to a runtime error. However, this is ruled out by preservation (Theorem 5) of System Ξ. Furthermore, the translation does not introduce any delimiters or uses runtime labels in any other way.

2.5 Related Work

In this chapter we introduced Effekt, a language with lexical effect handlers, and System Ξ, a language in explicit capability-passing style. Effect safety is guaranteed by all capabilities and blocks being second class. In this section we compare to work on dynamic effect handlers, lexical effect handlers, capability-passing style, and second-class values.

2.5.1 Dynamic Effect Handlers

In languages with *dynamic effect handlers*, effect operations are handled by the dynamically closest handler [Plotkin and Pretnar, 2013, Dolan et al., 2014, Hillerström and Lindley, 2016, Leijen, 2017b, Lindley et al., 2017]. Operationally, they search the runtime stack for the first handler for the effect operation, whereas in our operational semantics we search the current runtime stack for the first matching label. Consequently, these languages have a different type-and-effect system, usually based on effect rows [Leijen, 2017b, Lindley et al., 2017, Hillerström and Lindley, 2016] whereas thanks to our restriction to second-class blocks we don't need an effect system at all. To prevent accidental handling of effect operations, and to encapsulate effects, they feature term-level lifting constructs [Biernacki et al., 2017, Leijen, 2018, Convent et al., 2020, Saleh et al., 2018] whereas with explicit capability passing effect operations are lexically associated to their handler.

In summary, in comparison to these languages with dynamic effect handlers, Effekt uniquely guarantees effect safety, avoids accidental capture, and establishes effect encapsulation without lifts with a very lightweight effect system.

2.5.2 Lexical Effect Handlers

Our compilation technique with explicit capability-passing style implements *lexical effect handlers*.

The language Effekt that we presented in this chapter is based on a series of library implementations of lexical effect handlers published under the name Effekt [Brachthäuser and Schuster, 2017, Brachthäuser et al., 2018, 2020]. All of them are based on capability-passing style. They do not present a formal calculus but use capability passing in their library embeddings of effect handlers. To capture the continuation, they use a monadic implementation of multi-prompt delimited continuations [Dybvig et al., 2007]. Their capabilities are pairs of the handler implementation and a prompt. Passing capabilities explicitly facilitates optimizations by the JVM.

Brachthäuser and Schuster [2017] present a Scala [Odersky, 2019] library which reuses Scala's implicit parameters features to pass capabilities implicitly. It does not have an effect system and does not guarantee effect safety. Brachthäuser et al. [2018] present a Java [Gosling et al., 1996] library where capabilities are passed explicitly. It uses a bytecode transformation to continuation-passing style. Programs are written in direct style. It does not have an effect system and does not guarantee effect safety. Brachthäuser et al. [2020] present a Scala library where effect safety is guaranteed. It reuses Scala's path-dependent types and intersection types to track the set of capabilities each method uses.

With dynamic effect handlers, when a higher-order function uses its function argument under a handler, the effects of the function argument will be handled by this handler. This is not always desired. Zhang et al. [2016] observe this problem with Java checked exceptions. As a solution, they translate methods that throw exceptions to receive and pass along an additional parameter that will be a label at runtime. Their solution essentially makes exceptions lexically scoped. Zhang and Myers [2019] generalize this problem and its solution from exceptions to effect handlers. By establishing a lexical binding, lifting annotations are not required while guaranteeing effect parametric reasoning. Zhang et al. [2020] generalize lexical effect handlers to support bidirectional control flow. Bidirectional handlers can resume with a computation which will run in the context of the call-site of the effect operation.

Bauer and Pretnar [2015] explicitly pass effect instances. Without a static effect system, their language does not guarantee effect safety. In contrast to our capabilities, effect instances are first-class. Bauer and Pretnar [2013] present an effect system for a language with effect handlers. They assume a statically fixed set of effect instances, which are term-level constants. In some cases the instances a program uses are not known statically, and thus the handler cannot remove the instance from the effect type of the handled expression.

Biernacki et al. [2019b] present Helium, a language with lexical effect handlers and effect safety similar to the one in [Zhang and Myers, 2019]. They argue that lexically scoped effects improve reasoning. Explicitly binding effects also allows to refer to one particular effect instance in the presence of multiple copies of the same effect. Their operational semantics does not employ capability passing as they look up handler implementations based on a label when an effect operation is called. They use multi-prompt delimited control to get access to the current continuation. To guarantee effect safety, they index functions with sets of runtime labels. To relieve the programmer from passing effect instances manually, the implementation of Helium sometimes does this passing implicitly, but they do not describe when and how this happens.

2.5.3 Second class values

Effect safety of System Ξ and consequently of Effekt rests on the restriction of blocks to be second class. Hannan [1998] were the first to present a type-based escape analysis. They distinguish between functions whose parameters are first class and functions whose parameters are second class with two different types. In contrast to this, in Effekt parameters of value type are always first class and parameters of block type are always second class. Osvald et al. [2016] directly inspired our treatment of functions as second-class values. They use second class functions to establish a type-based escape analysis and present case studies for exceptions, memory regions, and well-scopedness in program generation. We generalize their calculus to effect handlers, and present a language design around this insight.

2.6 Conclusion

In this chapter we have introduced Effekt, a language with lexical effect handlers. Effect types express which capabilities a computation requires from its context. We have presented System Ξ a language with lexical effect handlers in explicit capability-passing style, and a translation of Effekt to System Ξ. While programming in Effekt is convenient, the flow of capabilities is explicit in System Ξ. In the rest of this thesis we will build on and exploit this explicitness.

All blocks in Effekt and in System Ξ are restricted to be second class. This makes it possible to guarantee effect safety without any effect system and the associated ceremony. We believe that the vast majority of programs uses functions and capabilities in a second-class way. However, in the future we would like to investigate how to lift this restriction and offer first-class functions in a language like Effekt. It is important to us, though, that in the case where functions and capabilities are used in a second-class way the ergonomics are retained.

3 Continuation Passing

Abstract. In Chapter 2 we have seen that effect handlers get access to the current continuation. They are useful to express complex control flow, a property they share with classical control operators like `shift` and `shift0`. Effect handlers are nested. Each handler introduces a delimiter and operationally effect operations capture the current continuation up to this delimiter.

In this chapter we revisit and combine classical work on continuation-passing style. Translating programs to continuation-passing style (CPS) is a well-known implementation technique for control operators. To account for the nesting of delimiters, we translate programs into the CPS hierarchy, where functions receive not one but potentially multiple continuations corresponding to multiple delimiters. While the original introduction of the CPS hierarchy works with an untyped language, in this chapter we present an implementation of the CPS hierarchy as a typed embedding into a dependently-typed language. We index effectful terms by the list of answer types they require from their context: the stack shape. Each type in this list corresponds to one level of the CPS hierarchy. We also use types to distinguish between two stages: static and dynamic. Our translation avoids administrative beta- and eta-redexes at all levels of the CPS hierarchy, by iterating well-known techniques for the non-iterated CPS translation.

Iterated continuation passing is the second ingredient of the compilation technique presented in this thesis.

In Chapter 4 we will account for effect-polymorphic functions by abstracting over type-level regions and term-level subregion evidence. Operationally, subregion evidence lifts a computation to run in a different context with a larger stack shape. We show that typed lexical effect handlers do not need the full power of multi-prompt delimited control, but can use iterated continuation passing instead.

In Chapter 5 we demonstrate how the combination of iterated continuation passing and explicit capability passing allows us to eliminate handler abstractions. Being explicit about continuations allows us to specialize uses of effect operations to their surrounding context.

This chapter is based on the following publication: Philipp Schuster and Jonathan Immanuel Brachthäuser. 2018. “Typing, Representing, and Abstracting Control.” *Proceedings of the Workshop on Type-Driven Development*, (TyDe). DOI: https://doi.org/10.1145/3240719.3241788

```
fail : STM (String :: RS) A
fail = SHIFT0 (λk ⇒ do
  PURE "no")

fork : STM (R :: RS) Bool
fork = SHIFT0 (λk ⇒ do
  RESUME k True
  RESUME k False)

emit : A → STM (List A :: RS) Unit
emit a = SHIFT0 (λk ⇒ do
  as ← RESUME k ()
  PURE (Con a as))
```

Figure 3.1: Examples of effectful functions.

3.1 Programming with Control Operators

In this section, we motivate programming with control operators in the CPS hierarchy and give an overview over our source language as an embedding into the dependently typed programming language Idris [Brady, 2013].

3.1.1 Example: Non-Deterministic Programming

We start with an example implementation of the non-deterministic programming statements fail and fork in terms of the control operator SHIFT0 in Figure 3.1, adopted from Danvy and Filinski [1990]. It might be instructive to compare with the original presentation. Both implementations capture the current continuation k with SHIFT0. In the implementation of fail we never resume and immediately return the string "no" instead. We do so with PURE, which embeds a pure value into a computation. In the implementation of fork we resume the continuation with both True and False. We write our programs in do-notation to sequence effectful statements: Each line after the **do** is a statement. We explicitly resume continuations with RESUME. The blue overbars and red underbars are staging annotations and can safely be ignored for now – we will explain them later.

The types of both fork and fail show that they are (effectful) statements STM. Statements are parametrized by the stack shape and their immediate result. The stack shape is the list of answer types at enclosing delimiters from innermost to outermost. It intuitively corresponds to the computational context that needs to be provided to execute the statement. Every element in the list marks a position of the runtime stack with the expected type. We will use these positions as targets to transfer the control-flow to. The immediate result of fail is A for all A, expressing that fail never returns anything (it could be the empty type Void). To allow for aborting the computation with the string "no", fail requires the stack shape to start with String. In contrast, fork is both polymorphic in its (top-most) answer type R and the rest of the answer types RS. It captures the continuation using SHIFT0 and resumes it twice, discarding the first result of type R. It executes both alternatives for their side effects. The immediate result of fork is Bool, hence the continuation k takes a boolean value to resume execution.

Because SHIFT0 is a *delimited* control operator it will capture the continuation only up to the closest delimiter RESET. In consequence, the following example will return the string "Answer was : no" and not terminate the execution as a whole to return with the string "no".

```
delimitFail : STM RS String
delimitFail = do
  a ← RESET fail
  PURE (concat "Answer was : " a)
```

Here we delimit the statement fail with RESET. Thus, fail will not discard the entire continuation but only up to the closest delimiter RESET. Since RESET immediately surrounds fail, the captured continuation happens to be empty. The variable a will be bound to the string "no" which is the immediate result of the delimited computation RESET fail. The type of the immediate result fits with the top-most answer type of fail whose stack shape is String :: RS.

In contrast, the following example will not type check:

```
perhapsFail : STM (String :: RS) Int
perhapsFail = ifthenelse (18 < 0) fail (PURE 9)

wontTypecheck : STM RS String
wontTypecheck = do
  a ← RESET perhapsFail
  PURE (concat "Answer was : " a)
```

It is not clear (to the compiler) whether the computation perhapsFail delimited by RESET will abort with string "no" or return normally with integer 9. When we delimit a statement with RESET the answer type and the immediate result type have to agree.

As another example, consider the effectful function emit in Figure 3.1 that yields a value to the surrounding context. It takes a value to emit as its argument a. It uses SHIFT0 to capture the current continuation k and resumes k with the unit value to get a list of results as. Finally, it prepends its argument a to the front of the other results as. Since we expect resuming the continuation to return a list, the effectful function emit only works when the top-most answer type is a list.

To gather the list of emitted values we define the function gather:

```
gather : STM (List A :: RS) B → STM RS (List A)
gather prog = RESET (do
  prog
  PURE Nil)
```

The function gather runs a computation that has List A as its top-most answer type and an arbitrary result type B. It runs the computation, ignores its result and then

```
choice : Int → STM (String :: RS) Int
choice = defrec (λrecurse ⇒ λn ⇒ do
  ifthenelse (n ≤ 1)
    fail
    (do
      b ← fork
      ifthenelse b
        (recurse (n − 1))
        (PURE n)))

triple : Int → Int → STM (String :: RS) (Int, Int, Int)
triple n m = do
  a ← choice n
  b ← choice (a − 1)
  c ← choice (b − 1)
  ifthenelse ((a + b + c) ≡ m)
    (PURE (a , b , c))
    fail
```

Figure 3.2: Effectful functions for choosing an integer, and finding triples.

returns the empty list. It also acts as a delimiter for all calls to SHIFT0 in prog such as the one in emit. In consequence, emit will suspend and resume the computation at the surrounding call to gather, prepending the emitted values to the empty list after the delimited continuation returns.

3.1.2 Example: Collecting Triples

We now present a bigger example also adopted from Danvy and Filinski [1990]. The task is to generate all distinct positive integers a, b and c less than or equal to a given integer n that sum to a given integer m. Our strategy is to use the non-deterministic choice statement fork to generate candidate triples and the abortive statement fail to filter those that do not have the desired property.

Equipped with the operators fork and fail, in Figure 3.2 we define a recursive function choice that non-deterministically chooses an integer between 1 and a given integer n. We then use choice to define a function triple that chooses integers a, b and c and fails if they do not sum up to the given integer m. To be able to use fail, the types of choice and triple must express that they use control effects with top-most answer type String, but are polymorphic in the remaining answer types RS.

To gather all triples produced by triple into a list, we might want to use emit and gather. However, when trying to do so, we already notice on the type-level, that the two effects interfere. We cannot have a top-most answer type of both String and of List (Int, Int, Int) at the same time. The solution is to introduce a second level of control and use the emit operation on that level. To do so, we use the operator LIFT that lifts a computation from one level to the next. We will define it later.

```
emitTriples : STM (String :: List (Int, Int, Int) :: RS) String
emitTriples = do
  res ← triple 9 15
  LIFT (emit res)
  PURE "done"

emittedTriples : STM • (List (Int, Int, Int))
emittedTriples = gather (RESET emitTriples)
```

In emittedTriples, we choose the stack shape RS to be the empty list •. As a result, emittedTriples cannot have any further control effects itself – making it a pure expression. The effects of emitTriples (that is, those of choice and fail) are delimited by RESET and the effects of emit are delimited by gather.

An alternative to collecting all triples is to abort the computation early and only get the first triple. To this end, we define an effectful function first that, when called with an argument a, gets the current continuation, but never resumes it. Instead it immediately returns Just a.

```
first : A → STM (Maybe A :: RS) ()
first a = SHIFT0 (λk ⇒ do
  PURE (Just a))
```

Running triple with first instead of emit follows the same pattern as before.

```
firstOfTriples : STM (String :: Maybe (Int, Int, Int) :: RS) String
firstOfTriples = do
  res ← triple 9 15
  LIFT (first res)
  PURE "done"

firstTriple : STM • (Maybe (Int, Int, Int))
firstTriple = RESET (do
  RESET firstOfTriples
  PURE Nothing)
```

The effectful function first requires an answer type of Maybe A. We can use it on the result of triple by lifting it, just like we did before with emit. However, we now discard the result of RESET firstTriples and return Nothing whenever triple fails to return a value.

Effect handlers allow us to abstract over the difference between emittedTriples and firstTriple. The effectful program defines the search space using fork and fail and the handler defines the strategy used to explore this space.

Figure 3.3 shows the result of pretty printing the program that we generate from `emittedTriples`. All control operators are eliminated and evaluating the expression

```
letrec f0 n = \k1 ⇒ \k2 ⇒
  if (n < 1)
    then k2 "no"
    else f0 (n - 1) k1 (\x4 ⇒ k1 n k2)
in f0 9 (\x0 ⇒ \k3 ⇒
  letrec f2 n = \k1 ⇒ \k2 ⇒
    if (n < 1)
      then k2 "no"
      else f2 (n - 1) k1 (\x6 ⇒ k1 n k2)
  in f2 (x0 - 1) (\x1 ⇒ \k4 ⇒
    letrec f4 n = \k1 ⇒ \k2 ⇒
      if (n < 1)
        then k2 "no"
        else f4 (n - 1) k1 (\x8 ⇒ k1 n k2)
    in f4 (x1 - 1) (\x2 ⇒ \k5 ⇒
          if ((x0 + x1 + x2) == 15)
            then (Con (x0, x1, x2) (k5 "done"))
            else k5 "no") k4) k3) (\x0 ⇒ Nil)
```

Figure 3.3: Result of pretty printing the expression `emittedTriples`.

results in a pure list of triples. It is specialized to use two levels of CPS internally. The CPS transformation only introduced beta-redexes in-between recursive definitions. The resulting program is free of administrative beta- and eta-redexes. It consists of three nested loops.

We have shown how to use our embedded language to compose programs with control operators. Since our embedding is typed, we were able to prevent some errors and for instance reject the function wontTypecheck. We have shown and discussed the code in CPS that we generate. In the next section we start building up the necessary machinery with an easy first step.

3.2 Basics: Continuation-Passing Style

Our goal is to embed a language with control operators into the dependently typed language Idris. In this section, we start with the translation of control operators SHIFT0 and RESET into CPS and show how to enable do-notation for terms in CPS.

We introduce the following type alias to represent terms in CPS with return type A and answer type R.

$$\textsc{Cps} : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}$$
$$\textsc{Cps}\,\mathsf{R}\,\mathsf{A} = (\mathsf{A} \to \mathsf{R}) \to \mathsf{R}$$

We can also view a term of type $\textsc{Cps}\,\mathsf{R}\,\mathsf{A}$ as potentially having control effects up to a delimiter that expects type R. In this section, a context from type A to type R is a function $(\mathsf{A} \to \mathsf{R})$. We will use the terms context and continuation interchangeably.

Following Materzok and Biernacki [2011], we define the control operator Shift0. Given a body, it returns a term in CPS. The body takes the captured continuation from immediate result type A to answer type R and returns R.

$$\begin{aligned}&\textsc{Shift0} : ((\mathsf{A} \to \mathsf{R}) \to \mathsf{R}) \to \textsc{Cps}\,\mathsf{R}\,\mathsf{A}\\ &\textsc{Shift0}\,\mathsf{m} = \mathsf{m}\end{aligned}$$

We define Shift0 to be the identity function - it is just a shift in perspective so to say. It is important to note that Shift0 removes the corresponding Reset delimiter and that both the continuation and the body of Shift0 have to be pure. Neither of them can have any control effects as those would be undelimited. This is also reflected in the type of the body and the continuation: both return a pure value of type R.

The control operator Shift0 also has an inverse that we will call RunIn0. It is the dollar operator from Kiselyov and Shan [2007] but with its arguments swapped. Given a term in CPS and a context, it runs the term in the context, delimiting any control effects the term might have.

$$\begin{aligned}&\textsc{RunIn0} : \textsc{Cps}\,\mathsf{R}\,\mathsf{A} \to (\mathsf{A} \to \mathsf{R}) \to \mathsf{R}\\ &\textsc{RunIn0}\,\mathsf{m} = \mathsf{m}\end{aligned}$$

The result of RunIn0 is a pure value of type R, it does not have any control effects. We will later see how Shift0 and RunIn0 make it very natural to walk up and down the CPS hierarchy.

To recover the classical delimiter Reset we run the computation in the empty context, which is the identity function.

$$\begin{aligned}&\textsc{Reset} : \textsc{Cps}\,\mathsf{A}\,\mathsf{A} \to \mathsf{A}\\ &\textsc{Reset}\,\mathsf{m} = \textsc{RunIn0}\,\mathsf{m}\,(\lambda \mathsf{x} \Rightarrow \mathsf{x})\end{aligned}$$

The return type A and the answer type need to agree. Again, the type makes it clear that Reset delimits all control effects. The result is a pure value.

To translate pure values into CPS we define the function Pure and to compose terms in CPS we define the function Bind. The reader might recognize the continuation monad. And indeed, these are the two combinators that allow us to use do-notation for our embedded language.

$$\begin{aligned}&\textsc{Pure} : \mathsf{A} \to \textsc{Cps}\,\mathsf{R}\,\mathsf{A}\\ &\textsc{Pure}\,\mathsf{a} = \lambda \mathsf{k} \Rightarrow \mathsf{k}\,\mathsf{a}\end{aligned}$$

$$\begin{aligned}&\textsc{Push} : (\mathsf{A} \to \textsc{Cps}\,\mathsf{R}\,\mathsf{B}) \to (\mathsf{B} \to \mathsf{R}) \to (\mathsf{A} \to \mathsf{R})\\ &\textsc{Push}\,\mathsf{f}\,\mathsf{k} = \lambda \mathsf{a} \Rightarrow \mathsf{f}\,\mathsf{a}\,\mathsf{k}\end{aligned}$$

$$\begin{aligned}&\textsc{Bind} : \textsc{Cps}\,\mathsf{R}\,\mathsf{A} \to (\mathsf{A} \to \textsc{Cps}\,\mathsf{R}\,\mathsf{B}) \to \textsc{Cps}\,\mathsf{R}\,\mathsf{B}\\ &\textsc{Bind}\,\mathsf{m}\,\mathsf{f} = \lambda \mathsf{k} \Rightarrow \mathsf{m}\,(\mathsf{push}\,\mathsf{f}\,\mathsf{k})\end{aligned}$$

The operator PURE calls the current continuation with the given value. We define an auxiliary function PUSH to push an effectful function $(\mathsf{A} \to \textsc{Cps}\,\mathsf{R}\,\mathsf{B})$ onto a context $(\mathsf{B} \to \mathsf{R})$ to get a new context $(\mathsf{A} \to \mathsf{R})$. We call it "push" to emphasize the analogy between computational contexts and the runtime stack. In BIND we run the given term in CPS with the given effectful function pushed onto the current continuation.

Let us summarize what we have so far in a small example:

```
example : Int
example = 1 + RESET (do
  x ← SHIFT0 (λk ⇒ k (k 100))
  PURE (10 + x))
```

The example evaluates to 121. With RESET we delimit the effectful term to get a value of type Int and add the value 1. In the argument of RESET we use our control operator SHIFT0 to capture the current continuation as k and apply it twice to the value 100. Like all the continuations that we capture with SHIFT0, k is pure. We bind the result of SHIFT0 to x and return this result after adding 10. Again, the type of example tells us that it is pure i.e. it does not have any control effects observable from the outside. All side effects have been encapsulated.

3.3 Representing Control: Staging CPS Expressions

In the previous section, we embedded the control operators SHIFT0 and RUNIN0 into our meta language Idris. Now we want to reify terms in the embedded language into our target language: a typed embedding of lambda calculus. In Section 3.6, we will then extend the target language with primitive operations, defrec and ifthenelse. The data type `Exp` represents an expression in lambda calculus in HOAS [Pfenning and Elliot, 1988] which simplifies the implementation considerably. This was observed before in a similar setting by Thiemann [1996].

```
data Exp : Type → Type where
  Lam : (Exp a → Exp b) → Exp (a → b)
  App : Exp (a → b) → Exp a → Exp b
```

We write type applications of data type `Exp` with a red underbar, as in $\underline{\mathsf{A}}$, to represent a target language expression of type A. For example, a term in the target language with function type has type $\underline{(\mathsf{A} \to \mathsf{B})}$, but a function in the meta language between expressions in the target language has type $\underline{\mathsf{A}} \to \underline{\mathsf{B}}$. Additionally, we write an application of the one-argument constructor `Lam` to a function as $\underline{\lambda}\mathsf{x}\,\underline{\Rightarrow}$ and the two-argument constructor `App` infix as $\underline{@}$.

We follow Danvy and Filinski [1992] and add staging annotations to the type of terms in CPS, that we have introduced in Section 3.2:

$$\overline{\textsc{Cps}} : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}$$
$$\overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{A} = (\underline{\mathsf{A}} \to \underline{\mathsf{R}}) \to \underline{\mathsf{R}}$$

We have two stages: present and future. We also call the present stage *static* and the future stage *dynamic* and use types to distinguish terms in different stages. Present-stage values of type A have type A and future-stage values of type A have type $\underline{\mathsf{A}}$.

Notational Conventions The naming convention of writing a type in blue and with an overbar (like $\overline{\textsc{Cps}}$) indicates that we statically know the control flow of a term. In contrast, a red type with an underbar is the type of an expression in the target language, which means that the term will only be known dynamically. A black type is a type in the meta language Idris. Similarly, on the term level, we have a naming convention where we use red with an underbar for constructors of target language expressions and black for terms in the meta language Idris.

In this chapter we conflate types and terms in the present stage with types and terms in the meta language to reduce notational overhead. In Chapter 5 we use a more precise notation.

We are ready to define staged variants of Shift0, RunIn0 and Reset. The definitions are exactly the same as in Section 3.2, however, using staging annotations we can give them more specific types. The types express the fact that we are composing (dynamic) target language expressions.

$$\begin{aligned}
&\textsc{Shift0} : ((\underline{\mathsf{A}} \to \underline{\mathsf{R}}) \to \underline{\mathsf{R}}) \to \overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{A}\\
&\textsc{Shift0}\,\mathsf{m} = \mathsf{m}
\end{aligned}$$

$$\begin{aligned}
&\textsc{RunIn0} : \overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{A} \to (\underline{\mathsf{A}} \to \underline{\mathsf{R}}) \to \underline{\mathsf{R}}\\
&\textsc{RunIn0}\,\mathsf{m} = \mathsf{m}
\end{aligned}$$

$$\begin{aligned}
&\textsc{Reset} : \overline{\textsc{Cps}}\,\mathsf{A}\,\mathsf{A} \to \underline{\mathsf{A}}\\
&\textsc{Reset}\,\mathsf{m} = \textsc{RunIn0}\,\mathsf{m}\,(\lambda\mathsf{x} \Rightarrow \mathsf{x})
\end{aligned}$$

To compose programs in CPS, we define staged variants of Pure and Bind with auxiliary function Push just like in Section 3.2. Again, on the term level they are exactly the same, but we give them more specific types.

$$\begin{aligned}
&\textsc{Pure} : \underline{\mathsf{A}} \to \overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{A}\\
&\textsc{Pure}\,\mathsf{a} = \lambda\mathsf{k} \Rightarrow \mathsf{k}\,\mathsf{a}
\end{aligned}$$

$$\begin{aligned}
&\textsc{Push} : (\underline{\mathsf{A}} \to \overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{B}) \to (\underline{\mathsf{B}} \to \underline{\mathsf{R}}) \to (\underline{\mathsf{A}} \to \underline{\mathsf{R}})\\
&\textsc{Push}\,\mathsf{f}\,\mathsf{k} = \lambda\mathsf{a} \Rightarrow \mathsf{f}\,\mathsf{a}\,\mathsf{k}
\end{aligned}$$

$$\begin{aligned}
&\textsc{Bind} : \overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{A} \to (\underline{\mathsf{A}} \to \overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{B}) \to \overline{\textsc{Cps}}\,\mathsf{R}\,\mathsf{B}\\
&\textsc{Bind}\,\mathsf{m}\,\mathsf{f} = \lambda\mathsf{k} \Rightarrow \mathsf{m}\,(\textsc{Push}\,\mathsf{f}\,\mathsf{k})
\end{aligned}$$

Expressions that are built using the above functions are meta level functions over terms of the target language. Eventually, we want to completely reify such expressions to one expression in the target language. To this end, we define the two symmetric functions REIFY and REFLECT:

$$\begin{array}{l}
\textsc{Reify} : \overline{\textsc{Cps}}\,\mathsf{R\,A} \rightarrow \underline{\textsc{Cps}\,\mathsf{R\,A}}\\
\textsc{Reify}\,\mathsf{m} = \underline{\lambda \mathsf{k} \Rightarrow}\,\mathsf{m}\,(\lambda\mathsf{a} \Rightarrow \mathsf{k}\,\underline{@}\,\mathsf{a})
\end{array}$$

$$\begin{array}{l}
\textsc{Reflect} : \underline{\textsc{Cps}\,\mathsf{R\,A}} \rightarrow \overline{\textsc{Cps}}\,\mathsf{R\,A}\\
\textsc{Reflect}\,\mathsf{m} = \lambda\mathsf{k} \Rightarrow \mathsf{m}\,\underline{@}\,(\underline{\lambda\mathsf{a} \Rightarrow}\,\mathsf{k\,a})
\end{array}$$

We naturally avoid any administrative beta redexes that would have to be post-reduced as explained by Danvy and Filinski [1992]. To translate a term in a dynamic context k, the function reify takes a term where the control flow is known statically but values are only known dynamically and produces a term in the target language. The function reflect takes a term in the target language and makes it possible to use it in the source language.

Given the previous definitions, we can now embed application and abstraction. Application takes a term in CPS whose result is an effectful function and a term in CPS whose result is a value of type A and applies the function to the value. This makes it necessary to first evaluate both the function and the value and finally reflect the result of the application. To embed lambda abstractions we take a function (again in HOAS) and return an abstraction in the target language where we reify the body of the function applied to the argument.

$$\begin{array}{l}
\mathsf{apply} : \overline{\textsc{Cps}}\,\mathsf{R}\,(\mathsf{A} \rightarrow \textsc{Cps}\,\mathsf{R\,B}) \rightarrow \overline{\textsc{Cps}}\,\mathsf{R\,A} \rightarrow \overline{\textsc{Cps}}\,\mathsf{R\,B}\\
\mathsf{apply\,mf\,ma} = \mathbf{do}\\
\quad \mathsf{f} \leftarrow \mathsf{mf}\\
\quad \mathsf{a} \leftarrow \mathsf{ma}\\
\quad \textsc{Reflect}\,(\mathsf{f}\,\underline{@}\,\mathsf{a})
\end{array}$$

$$\begin{array}{l}
\mathsf{lambda} : (\underline{\mathsf{A}} \rightarrow \overline{\textsc{Cps}}\,\mathsf{R\,B}) \rightarrow \overline{\textsc{Cps}}\,\mathsf{R}\,(\mathsf{A} \rightarrow \textsc{Cps}\,\mathsf{R\,B})\\
\mathsf{lambda\,f} = \textsc{Pure}\,(\underline{\lambda\mathsf{a} \Rightarrow}\,\textsc{Reify}\,(\mathsf{f\,a}))
\end{array}$$

These definitions coincide with the ones by Danvy and Filinski [1992]. All we did was to inline the translation meta-function and factor parts into reusable combinators.

Let's consider the example term from Section 3.2, but with the types presented in this section.

$$\begin{array}{l}
\mathsf{example} : \underline{\mathsf{Int}}\\
\mathsf{example} = \underline{1}\,\underline{+}\,(\textsc{Reset}\,(\mathbf{do}\\
\quad \mathsf{x} \leftarrow \textsc{Shift0}\,(\lambda\mathsf{k} \Rightarrow \mathsf{k}\,(\mathsf{k}\,\underline{100}))\\
\quad \textsc{Pure}\,(\underline{10}\,\underline{+}\,\mathsf{x})))
\end{array}$$

When we pretty print the expression example we get the following:

```
1 + (10 + (10 + 100))
```

All control flow is statically known and does not appear in the result.

To make this example more interesting, let's abstract the subterm with the call to Shift0 into its own function resumeTwice.

```
resumeTwice : Cps Int (Int → Cps Int Int)
resumeTwice = lambda (λn ⇒ Shift0 (λk ⇒ k (k n)))

example′ : Int
example′ = 1 + (Reset (do
  x ← apply resumeTwice (Pure 100)
  Pure (10 + x))))
```

Pretty printing this term now yields:

```
1 + (\n ⇒ \k ⇒ k (k n)) 100 (\x ⇒ 10 + x)
```

With lambda we reify an effectful function into the target language. To use it in the source language with apply, we have to reflect it. This introduces beta-redexes.

In this section we refined the definitions of Section 3.2 by giving more specific types and thereby adding staging annotations. We reify terms written in CPS in the source language into terms in CPS in the target language. The generated lambda expressions are free of administrative beta-redexes unless we specifically ask for a term to be reified and reflect it later. In the next section, we will again refine the basic definitions of Section 3.2, but in a different way.

3.4 Abstracting Control: The CPS Hierarchy

In this section, we will explore a second variation of the basic definitions of Section 3.2. Instead of adding staging annotations, this time we follow Danvy and Filinski [1990] and iterate the CPS translation to obtain a hierarchy of control operators. Later, in Section 3.5, we will combine the extensions of this and the previous section. The present section only uses terms in the meta language and therefore you will not see any colors.

The CPS hierarchy [Danvy and Filinski, 1990, Kameyama, 2004, Biernacka et al., 2011, Materzok and Biernacki, 2012] allows us to use different control effects in the same program. We can obtain the CPS hierarchy by iterating the CPS transformation. Since the CPS transformation transforms both types and terms, we will have to iterate it on both types and terms. As a consequence, we will have multiple answer types, one for each iteration of the CPS transformation. Concretely this means that the answer type of the first CPS transformation is again a term in CPS, whose answer type is then again in CPS and so on. For example using our definition of Cps from Section 3.2, the type of a term of type A in CPS whose answer types are P, Q and R would have type Cps (Cps (Cps R Q) P) A. As a shorthand, Figure 3.4 defines the type of statements Stm RS A with a stack shape RS and a return type A. The stack shape is the list of answer types. Here we can see the power of using dependent types. We index statements by a list and thus statically track all intermediate answer types on the type level. We type members of the CPS hierarchy of level n at type Stm RS A

$\textsc{Stm} : \mathsf{List\,Type} \rightarrow \mathsf{Type} \rightarrow \mathsf{Type}$
$\textsc{Stm}\,(\bullet)\,\mathsf{A} = \mathsf{A}$
$\textsc{Stm}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A} = \textsc{Cps}\,(\textsc{Stm}\,\mathsf{RS}\,\mathsf{R})\,\mathsf{A}$

(a) Type of effectful statements, indexed by answer types.

$\textsc{Pure} : \mathsf{A} \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{A}$
$\textsc{Pure}_{(\mathsf{RS}=\bullet)}\,\mathsf{a} = \mathsf{a}$
$\textsc{Pure}_{(\mathsf{RS}=(\mathsf{Q}\,::\,\mathsf{QS}))}\,\mathsf{a} = \lambda\mathsf{k} \Rightarrow \mathsf{k}\,\mathsf{a}$

$\textsc{Push} : (\mathsf{A} \rightarrow \textsc{Stm}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{B}) \rightarrow (\mathsf{B} \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \rightarrow (\mathsf{A} \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{R})$
$\textsc{Push}\,\mathsf{f}\,\mathsf{k} = \lambda\mathsf{a} \Rightarrow \mathsf{f}\,\mathsf{a}\,\mathsf{k}$

$\textsc{Bind} : \textsc{Stm}\,\mathsf{RS}\,\mathsf{A} \rightarrow (\mathsf{A} \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{B}) \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{B}$
$\textsc{Bind}_{(\mathsf{RS}=\bullet)}\,\mathsf{m}\,\mathsf{f} = \mathsf{f}\,\mathsf{m}$
$\textsc{Bind}_{(\mathsf{RS}=(\mathsf{Q}\,::\,\mathsf{QS}))}\,\mathsf{m}\,\mathsf{f} = \lambda\mathsf{k} \Rightarrow \mathsf{m}\,(\textsc{Push}\,\mathsf{f}\,\mathsf{k})$

(b) Iterated variant of monadic operations.

$\textsc{Lift} : \textsc{Stm}\,\mathsf{RS}\,\mathsf{A} \rightarrow \textsc{Stm}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A}$
$\textsc{Lift} = \textsc{Bind}$

(c) Lift operation to move between layers of the hierarchy.

$\textsc{Shift0} : ((\mathsf{A} \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \rightarrow \textsc{Stm}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A}$
$\textsc{Shift0}\,\mathsf{m} = \mathsf{m}$

$\textsc{RunIn0} : \textsc{Stm}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A} \rightarrow (\mathsf{A} \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}$
$\textsc{RunIn0}\,\mathsf{m} = \mathsf{m}$

$\textsc{Reset} : \textsc{Stm}\,(\mathsf{A} :: \mathsf{RS})\,\mathsf{A} \rightarrow \textsc{Stm}\,\mathsf{RS}\,\mathsf{A}$
$\textsc{Reset}\,\mathsf{m} = \textsc{RunIn0}\,\mathsf{m}\,\textsc{Pure}$

(d) Iterated variant of control operations.

Figure 3.4: Iterated variant.

for a stack shape with length `n`. A statement with an empty stack shape cannot have any control effects and is a pure value. Because different parts of a program can have different stack shapes this will allow us to avoid CPS transforming sub-programs when it is unnecessary.

Again, Figure 3.4 implements SHIFT0, RUNIN0 and PUSH exactly like in Section 3.2. We just give them more specific types, replacing R by STM RS R. However, for PURE and BIND (Figure 3.4) we now have two cases to consider: one where we have an empty stack shape and one where we have a non-empty stack shape. For an empty stack shape i.e. pure expressions, in PURE we directly return the given value and in BIND we apply the second argument to the first. For a non-empty stack shape the implementation is exactly the one in Section 3.2. Furthermore, RESET does not use the identity function as empty context, but instead resets with PURE which corresponds to the function θ of Danvy and Filinski [1990].

The definitions given so far only work with the top-most answer type, but we want a hierarchy of control operators. While the CPS hierarchy is usually used to implement a family of control operators based on `shift` and `reset`, in this chapter we use it to construct a family of control operators SHIFT00, SHIFT01, SHIFT02, etc. based on SHIFT0 and RESET. Rather than defining them directly we will first define a useful function LIFT (Figure 3.4) that lifts any statement with answer types RS into a larger context with one more answer type R :: RS. Its implementation is just BIND and the reader might be surprised that the types just happen to match.

We obtain a family of shifts by iterating the lifting:

$$
\begin{aligned}
&\textsc{Shift00} : ((\mathsf{A} \to \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \to \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \to \textsc{Stm}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A}\\
&\textsc{Shift00}\,\mathsf{m} = \textsc{Shift0}\,\mathsf{m}
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{Shift01} : ((\mathsf{A} \to \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \to \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \to \textsc{Stm}\,(\mathsf{Q} :: \mathsf{R} :: \mathsf{RS})\,\mathsf{A}\\
&\textsc{Shift01}\,\mathsf{m} = \textsc{Lift}\,(\textsc{Shift0}\,\mathsf{m})
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{Shift02} : ((\mathsf{A} \to \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \to \textsc{Stm}\,\mathsf{RS}\,\mathsf{R}) \to \textsc{Stm}\,(\mathsf{P} :: \mathsf{Q} :: \mathsf{R} :: \mathsf{RS})\,\mathsf{A}\\
&\textsc{Shift02}\,\mathsf{m} = \textsc{Lift}\,(\textsc{Lift}\,(\textsc{Shift0}\,\mathsf{m}))
\end{aligned}
$$

The body of each SHIFT0i has the same type, since the control operator SHIFT0i removes i + 1 delimiters. It thus can only make use of control effects outside of the (i + 1)th delimiter. This also becomes visible in the result type: the answer type R has to match the answer type at the corresponding level of the outer computation. For example in SHIFT01, the answer type R occurs in the second position in the stack shape Q :: R :: RS.

Similarly, we can iterate RESET to obtain a family of resets that delimits multiple levels of control effects at once.

$$
\begin{aligned}
&\textsc{Reset1} : \textsc{Stm}\,(\mathsf{A} :: \mathsf{A} :: \mathsf{RS})\,\mathsf{A} \to \textsc{Stm}\,\mathsf{RS}\,\mathsf{A}\\
&\textsc{Reset1}\,\mathsf{m} = \textsc{Reset}\,(\textsc{Reset}\,\mathsf{m})
\end{aligned}
$$

All answer types have to agree.

Equipped with LIFT and RESET we can recover the classical control operator **shift**.

```
SHIFT : ((A → STM (R :: RS) R) → STM (R :: RS) R) → STM (R :: RS) A
SHIFT body = SHIFT0 (λk ⇒ RESET (body (λx ⇒ LIFT (k x))))
```

The difference shows in the type signature for SHIFT where the body and continuation live at the same level of the hierarchy as the rest. To implement SHIFT, we capture the continuation with SHIFT0, but delimit the body with RESET. Since the body now can have the same control effects, we also lift the continuation before passing it to the body.

An example of using multiple levels of control effects is the partition function. Given an integer a, it partitions a list of integers into two lists: one containing all integers less than a and on containing all integers greater or equal to a. We use emit on two levels of the hierarchy to emit values to the respective partition.

```
partition : Int → List Int → STM [List Int, List Int] Unit
partition a list = case list of
  Nil ⇒ do
    PURE ()
  Con hd tl ⇒ if (a < hd)
    then (do
      emit hd
      partition a tl)
    else (do
      LIFT (emit hd)
      partition a tl)
```

Having seen how to walk down the hierarchy with RESET and walk it up with LIFT, in the next section we will combine Section 3.3 and Section 3.4 in order to reify terms in the CPS hierarchy.

3.5 Representing and Abstracting Control

Combining the two variations of the previous two sections, we add staging annotations to the CPS hierarchy. We use the same definition of expressions as before. Our type of statements with staging annotations now marks the immediate result as well as all intermediate answer types as dynamic by wrapping them in `Exp` (Figure 3.5). When compared to Section 3.4, for the monadic operations (Figure 3.5), control operators (Figure 3.5) and LIFT (Figure 3.5) we only change the types to be more specific – the implementation is exactly the same. All abstractions and applications that we introduce are on the meta level which means in using these functions we do not generate any beta redexes in the target language. The control flow in the CPS hierarchy is completely static.

Just like in Section 3.3, we want to reify terms. The only difference is, that the

$$
\begin{aligned}
&\overline{\textsc{Stm}} : \mathsf{List\ Type} \to \mathsf{Type} \to \mathsf{Type}\\
&\overline{\textsc{Stm}}\ \bullet\ \mathsf{A} = \underline{\mathsf{A}}\\
&\overline{\textsc{Stm}}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A} = (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R}) \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R}
\end{aligned}
$$

(a) Type of effectful statements, indexed by intermediate answer types.

$$
\begin{aligned}
&\textsc{Pure} : \underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{A}\\
&\textsc{Pure}_{(\mathsf{RS}=\bullet)}\ \mathsf{a} = \mathsf{a}\\
&\textsc{Pure}_{(\mathsf{RS}=(\mathsf{Q} :: \mathsf{QS}))}\ \mathsf{a} = \lambda\mathsf{k} \Rightarrow \mathsf{k}\,\mathsf{a}\\
\\
&\textsc{Push} : (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{B}) \to (\underline{\mathsf{B}} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R}) \to (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R})\\
&\textsc{Push}\,\mathsf{f}\,\mathsf{k} = \lambda\mathsf{a} \Rightarrow \mathsf{f}\,\mathsf{a}\,\mathsf{k}\\
\\
&\textsc{Bind} : \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{A} \to (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{B}) \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{B}\\
&\textsc{Bind}_{(\mathsf{RS}=\bullet)}\ \mathsf{m}\,\mathsf{f} = \mathsf{f}\,\mathsf{m}\\
&\textsc{Bind}_{(\mathsf{RS}=(\mathsf{Q} :: \mathsf{QS}))}\ \mathsf{m}\,\mathsf{f} = \lambda\mathsf{k} \Rightarrow \mathsf{m}\,(\textsc{Push}\,\mathsf{f}\,\mathsf{k})
\end{aligned}
$$

(b) Iterated and staged variant of monadic operations.

$$
\begin{aligned}
&\textsc{Lift} : \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{A} \to \overline{\textsc{Stm}}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A}\\
&\textsc{Lift} = \textsc{Bind}
\end{aligned}
$$

(c) Lift operation to move between layers of the hierarchy.

$$
\begin{aligned}
&\textsc{Shift0} : ((\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R}) \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R}) \to \overline{\textsc{Stm}}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A}\\
&\textsc{Shift0}\,\mathsf{m} = \mathsf{m}\\
\\
&\textsc{RunIn0} : \overline{\textsc{Stm}}\,(\mathsf{R} :: \mathsf{RS})\,\mathsf{A} \to (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R}) \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{R}\\
&\textsc{RunIn0}\,\mathsf{m} = \mathsf{m}\\
\\
&\textsc{Reset} : \overline{\textsc{Stm}}\,(\mathsf{A} :: \mathsf{RS})\,\mathsf{A} \to \overline{\textsc{Stm}}\,\mathsf{RS}\,\mathsf{A}\\
&\textsc{Reset}\,\mathsf{m} = \textsc{RunIn0}\,\mathsf{m}\,\textsc{Pure}
\end{aligned}
$$

(d) Iterated and staged variant of control operations.

Figure 3.5: Iterated and staged variant.

source level terms live in the CPS hierarchy at an arbitrary level.

$$\begin{aligned}
&\textsc{Reify} : \overline{\textsc{Stm}}\ \mathsf{RS}\ \mathsf{A} \rightarrow \underline{\textsc{Stm}\ \mathsf{RS}\ \mathsf{A}}\\
&\textsc{Reify}_{(\mathsf{RS}\,=\,\bullet)}\ \mathsf{m}\ =\ \mathsf{m}\\
&\textsc{Reify}_{(\mathsf{RS}\,=\,(\mathsf{Q}\ ::\ \mathsf{QS}))}\ \mathsf{m}\ =\ \underline{\lambda \mathsf{k} \Rightarrow}\ \textsc{Reify}_{\mathsf{QS}}\ (\mathsf{m}\ (\lambda \mathsf{a} \Rightarrow \textsc{Reflect}_{\mathsf{QS}}\ (\mathsf{k}\ \underline{@}\ \mathsf{a})))
\end{aligned}$$

$$\begin{aligned}
&\textsc{Reflect} : \underline{\textsc{Stm}\ \mathsf{RS}\ \mathsf{A}} \rightarrow \overline{\textsc{Stm}}\ \mathsf{RS}\ \mathsf{A}\\
&\textsc{Reflect}_{(\mathsf{RS}\,=\,\bullet)}\ \mathsf{m}\ =\ \mathsf{m}\\
&\textsc{Reflect}_{(\mathsf{RS}\,=\,(\mathsf{Q}\ ::\ \mathsf{QS}))}\ \mathsf{m}\ =\ \lambda \mathsf{k} \Rightarrow \textsc{Reflect}_{\mathsf{QS}}\ (\mathsf{m}\ \underline{@}\ (\underline{\lambda \mathsf{a} \Rightarrow}\ \textsc{Reify}_{\mathsf{QS}}\ (\mathsf{k}\,\mathsf{a})))
\end{aligned}$$

The functions Reify and Reflect are mutually recursive. To reify a pure statement we don't need to do anything. It already is an expression in the target language. If we reify a statement with at least one answer type, we build an abstraction for the continuation and recursively reify the body after passing the reflected continuation to it. The body is one level lower in the hierarchy. Symmetrically, to reflect a pure expression into a statement without any answer types we don't have to do anything. If the expression has at least one answer type, we abstract the continuation and reflect the body after passing the reified continuation to it.

For example, let's define a function that emits the same value on two levels of control.

$$\begin{aligned}
&\mathsf{emitTwice}\ :\ \underline{\mathsf{Int}} \rightarrow \overline{\textsc{Stm}}\ (\mathsf{List\,Int}\ ::\ \mathsf{List\,Int}\ ::\ \mathsf{RS})\ \mathsf{Unit}\\
&\mathsf{emitTwice\,a}\ =\ \mathbf{do}\\
&\quad \mathsf{emit\,a}\\
&\quad \textsc{Lift}\ (\mathsf{emit\,a})
\end{aligned}$$

We have made emitTwice polymorphic in the rest of the stack shape RS. In choosing RS before reification we choose the level of the CPS hierarchy the reified term is in. For example, when we choose RS to be the empty list •, reify and pretty print the function emitTwice we get:

```
(\a ⇒ \k1 ⇒ \k2 ⇒
  Con a (k1 () (\as ⇒ k2 (Con a as))))
```

If we choose RS to be the singleton list containing type unit [Unit], we obtain one more level of CPS:

```
(\a ⇒ \k1 ⇒ \k2 ⇒ \k3 ⇒
  k1 () (\as ⇒ \k4 ⇒
    k2 (Con a as) (\x1 ⇒ k4 x1)) (\as ⇒
      k3 (Con a as)))
```

With the abstraction `(\x1 ⇒ k4 x1)`, we have introduced an eta-redex that when reduced exposes another eta-redex.

We have shown how to combine the ideas of Section 3.3 and Section 3.4 to take terms in our source language that use control operators at different levels and generate terms in our target language in the CPS hierarchy. While we could post-reduce those eta-redexes, it is advisable to not generate them in the first place, which we will do in the next section. This is important since the size of fully expanded types in the CPS hierarchy is exponential in the number of levels.

3.6 Preventing Eta-Redexes

In the previous section, we generated code for terms in the CPS hierarchy. But the code generated for statements with multiple answer types contains eta-redexes as observed by Danvy and Filinski [1992]. They also propose a solution to this problem: they distinguish whether the context is dynamic or static, avoiding unnecessary reflection and later reification of an already dynamic context. We can easily translate their solution to the iterated setting by distinguishing for every context in the CPS hierarchy whether it is static or dynamic. This will be our final version of the translation.

In Figure 3.6, we introduce a type of contexts $\overline{\textsc{Ctx}}\ \mathsf{RS}\ \mathsf{A}\ \mathsf{B}$ that corresponds to an effectful function $\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\ \mathsf{RS}\ \mathsf{B}$ from A to B with answer types RS. The two variants of type $\overline{\textsc{Ctx}}$ let us statically distinguish between static and dynamic contexts. A static context is an effectful function, like in Section 3.5. A dynamic context is an expression in the target language of type $\underline{(\mathsf{A} \to \textsc{Stm}\ \mathsf{RS}\ \mathsf{B})}$ where $\textsc{Stm}$ is from section 3.4. A statement now takes a context instead of an effectful function as its continuation. This makes our type of statements and our type of contexts mutually recursive.

With the distinction between static and dynamic context, Reify and Reflect are more complicated and use an auxiliary function ReifyCtx. Their definition is translated from [Danvy and Filinski, 1992], but where they have two mutually recursive functions that do the translation, we have two cases in ReifyCtx: one for static and one for dynamic continuations.

$$
\begin{aligned}
&\textsc{Reify} \;:\; \overline{\textsc{Stm}}\ \mathsf{RS}\ \mathsf{A} \to \underline{\textsc{Stm}\ \mathsf{RS}\ \mathsf{A}}\\
&\textsc{Reify}_{(\mathsf{RS}\,=\,\bullet)}\ \mathsf{m} \;=\; \mathsf{m}\\
&\textsc{Reify}_{(\mathsf{RS}\,=\,(\mathsf{Q}\ ::\ \mathsf{QS}))}\ \mathsf{m} \;=\; \underline{\lambda}\mathsf{k}\ \underline{\Rightarrow}\ \textsc{Reify}_{\mathsf{QS}}\ (\mathsf{m}\ (\mathsf{Dynamic}\ \mathsf{k}))
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{Reflect} \;:\; \underline{\textsc{Stm}\ \mathsf{RS}\ \mathsf{A}} \to \overline{\textsc{Stm}}\ \mathsf{RS}\ \mathsf{A}\\
&\textsc{Reflect}_{(\mathsf{RS}\,=\,\bullet)}\ \mathsf{m} \;=\; \mathsf{m}\\
&\textsc{Reflect}_{(\mathsf{RS}\,=\,(\mathsf{Q}\ ::\ \mathsf{QS}))}\ \mathsf{m} \;=\; \lambda\mathsf{k} \Rightarrow \textsc{Reflect}_{\mathsf{QS}}\ (\mathsf{m}\ \underline{@}\ (\textsc{ReifyCtx}\ \mathsf{k}))
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{ReifyCtx} \;:\; \overline{\textsc{Ctx}}\ \mathsf{RS}\ \mathsf{A}\ \mathsf{R} \to \underline{(\mathsf{A} \to \textsc{Stm}\ \mathsf{RS}\ \mathsf{R})}\\
&\textsc{ReifyCtx}\ (\mathsf{Static}\ \mathsf{k}) \;=\; \underline{\lambda}\mathsf{a}\ \underline{\Rightarrow}\ \textsc{Reify}\ (\mathsf{k}\ \mathsf{a})\\
&\textsc{ReifyCtx}\ (\mathsf{Dynamic}\ \mathsf{k}) \;=\; \mathsf{k}
\end{aligned}
$$

While in Section 3.5, in Reify we reflected the context before passing it to the statement m, we now wrap it in the $\mathsf{Dynamic}$ constructor. In Reflect, where we previously generated a lambda term to reify the context, we now call the function ReifyCtx instead. This will avoid generating a lambda abstraction when the context is dynamic.

In previous sections, continuations were functions and therefore we could apply them to a value to run them. Now we need a function Resume to run a continuation with a given value.

$$
\begin{aligned}
&\textsc{Resume} \;:\; \overline{\textsc{Ctx}}\ \mathsf{RS}\ \mathsf{A}\ \mathsf{R} \to (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\ \mathsf{RS}\ \mathsf{R})\\
&\textsc{Resume}\ (\mathsf{Static}\ \mathsf{k}) \;=\; \mathsf{k}\\
&\textsc{Resume}\ (\mathsf{Dynamic}\ \mathsf{k}) \;=\; \lambda\mathsf{a} \Rightarrow \textsc{Reflect}\ (\mathsf{k}\ \underline{@}\ \mathsf{a})
\end{aligned}
$$

Symmetrically to ReifyCtx, in Resume we reflect the context when it is dynamic but do nothing if it is static.

Once again, the definitions for Shift0 and RunIn0 do not change (Figure 3.6). Their types express that they now operate on contexts. In Pure, Push and Bind (Figure 3.6) we now have to resume the continuation with Resume instead of directly calling it. In Reset, we reset the computation into a context that is statically known to be Pure. Similarly, Push creates a static context.

With this definition of Bind we can't use do-notation anymore. So we define seq to call bind with its second argument wrapped in the `Static` constructor.

$$
\begin{aligned}
&\mathsf{seq}\ :\ \overline{\textsc{Stm}}\,\mathsf{RS\,A} \to (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS\,B}) \to \overline{\textsc{Stm}}\,\mathsf{RS\,B}\\
&\mathsf{seq\,m\,f}\ =\ \textsc{Bind}\,\mathsf{m}\,(\mathsf{Static\,f})
\end{aligned}
$$

Monadic composition in do-notation uses static contexts and still does not produce beta-redexes. In Reify and Reflect we take advantage of our ability to also pass dynamic contexts to statements to avoid eta-redexes.

3.6.1 Primitives, Branching and Recursion

Using this final version of our language, we now show how to add primitives, ifthenelse and defrec. In Section 3.3, we defined apply and lambda in the same way as Danvy and Filinski [1992]. However, it turns out that those definitions don't fit nicely with the rest of the language. We rather propose the following more symmetrical definitions.

$$
\begin{aligned}
&\mathsf{apply}\ :\ \underline{(\mathsf{A} \to \textsc{Stm}\,\mathsf{RS\,B})} \to (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS\,B})\\
&\mathsf{apply\,f}\ =\ \lambda\mathsf{a} \Rightarrow \textsc{Reflect}\,(\mathsf{f}\ \underline{@}\ \mathsf{a})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{lambda}\ :\ (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\,\mathsf{RS\,B}) \to \underline{(\mathsf{A} \to \textsc{Stm}\,\mathsf{RS\,B})}\\
&\mathsf{lambda\,f}\ =\ \underline{\lambda}\mathsf{a}\ \underline{\Rightarrow}\ \textsc{Reify}\,(\mathsf{f\,a})
\end{aligned}
$$

Here, the definition of apply does not take statements that evaluate to the function and the argument, respectively. Instead, it assumes they are already values and simply reflects the application. Likewise, the definition of lambda does not return a statement that then evaluates to the created lambda abstraction. Instead, the result of lambda is immediately the reified lambda abstraction.

In similar spirit, we do not CPS transform pure primitives like $\underline{+}$. While such a translation is possible as the following shows, we rather ask the user to explicitly use do-notation and bind.

$$
\begin{aligned}
&(+)\ :\ \overline{\textsc{Stm}}\,\mathsf{RS\,Int} \to \overline{\textsc{Stm}}\,\mathsf{RS\,Int} \to \overline{\textsc{Stm}}\,\mathsf{RS\,Int}\\
&(+)\,\mathsf{mx\,my}\ =\ \mathbf{do}\\
&\quad \mathsf{x} \leftarrow \mathsf{mx}\\
&\quad \mathsf{y} \leftarrow \mathsf{my}\\
&\quad \textsc{Pure}\,(\mathsf{x}\ \underline{+}\ \mathsf{y})
\end{aligned}
$$

This helps us to exploit the type-level distinction between pure expressions and effectful statements.

$$\begin{aligned}
&\overline{\textsc{Stm}} : \mathsf{List\ Type} \to \mathsf{Type} \to \mathsf{Type}\\
&\overline{\textsc{Stm}}\ \bullet\ \mathsf{A} = \underline{\mathsf{A}}\\
&\overline{\textsc{Stm}}\ (\mathsf{R} :: \mathsf{RS})\ \mathsf{A} = \overline{\textsc{Ctx}}\ \mathsf{RS\ A\ R} \to \overline{\textsc{Stm}}\ \mathsf{RS\ R}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{data}\ \overline{\textsc{Ctx}} : \mathsf{List\ Type} \to \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type\ where}\\
&\quad \mathsf{Static} : (\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\ \mathsf{RS\ B}) \to \overline{\textsc{Ctx}}\ \mathsf{RS\ A\ B}\\
&\quad \mathsf{Dynamic} : \underline{(\mathsf{A} \to \textsc{Stm}\ \mathsf{RS\ B})} \to \overline{\textsc{Ctx}}\ \mathsf{RS\ A\ B}
\end{aligned}$$

(a) Mutually recursive types of statements and contexts.

$$\begin{aligned}
&\textsc{Pure} : \underline{\mathsf{A}} \to \overline{\textsc{Stm}}\ \mathsf{RS\ A}\\
&\textsc{Pure}_{(\mathsf{RS}=\bullet)}\ \mathsf{a} = \mathsf{a}\\
&\textsc{Pure}_{(\mathsf{RS}=(\mathsf{Q} :: \mathsf{QS}))}\ \mathsf{a} = \lambda\mathsf{k} \Rightarrow \textsc{Resume}\ \mathsf{k\ a}
\end{aligned}$$

$$\begin{aligned}
&\textsc{Push} : \overline{\textsc{Ctx}}\ (\mathsf{R} :: \mathsf{RS})\ \mathsf{A\ B} \to \overline{\textsc{Ctx}}\ \mathsf{RS\ B\ R} \to \overline{\textsc{Ctx}}\ \mathsf{RS\ A\ R}\\
&\textsc{Push}\ \mathsf{f\ k} = \mathsf{Static}\ (\lambda\mathsf{a} \Rightarrow \textsc{Resume}\ \mathsf{f\ a\ k})
\end{aligned}$$

$$\begin{aligned}
&\textsc{Bind} : \overline{\textsc{Stm}}\ \mathsf{RS\ A} \to \overline{\textsc{Ctx}}\ \mathsf{RS\ A\ B} \to \overline{\textsc{Stm}}\ \mathsf{RS\ B}\\
&\textsc{Bind}_{(\mathsf{RS}=\bullet)}\ \mathsf{m\ f} = \textsc{Resume}\ \mathsf{f\ m}\\
&\textsc{Bind}_{(\mathsf{RS}=(\mathsf{Q} :: \mathsf{QS}))}\ \mathsf{m\ f} = \lambda\mathsf{k} \Rightarrow \mathsf{m}\ (\textsc{Push}\ \mathsf{f\ k})
\end{aligned}$$

(b) Iterated and staged variant of monadic operations.

$$\begin{aligned}
&\textsc{Lift} : \overline{\textsc{Stm}}\ \mathsf{RS\ A} \to \overline{\textsc{Stm}}\ (\mathsf{R} :: \mathsf{RS})\ \mathsf{A}\\
&\textsc{Lift} = \textsc{Bind}
\end{aligned}$$

(c) Lift operation to move between layers of the hierarchy.

$$\begin{aligned}
&\textsc{Shift0} : (\overline{\textsc{Ctx}}\ \mathsf{RS\ A\ R} \to \overline{\textsc{Stm}}\ \mathsf{RS\ R}) \to \overline{\textsc{Stm}}\ (\mathsf{R} :: \mathsf{RS})\ \mathsf{A}\\
&\textsc{Shift0}\ \mathsf{m} = \mathsf{m}
\end{aligned}$$

$$\begin{aligned}
&\textsc{RunIn0} : \overline{\textsc{Stm}}\ (\mathsf{R} :: \mathsf{RS})\ \mathsf{A} \to \overline{\textsc{Ctx}}\ \mathsf{RS\ A\ R} \to \overline{\textsc{Stm}}\ \mathsf{RS\ R}\\
&\textsc{RunIn0}\ \mathsf{m} = \mathsf{m}
\end{aligned}$$

$$\begin{aligned}
&\textsc{Reset} : \overline{\textsc{Stm}}\ (\mathsf{A} :: \mathsf{RS})\ \mathsf{A} \to \overline{\textsc{Stm}}\ \mathsf{RS\ A}\\
&\textsc{Reset}\ \mathsf{m} = \textsc{RunIn0}\ \mathsf{m}\ (\mathsf{Static}\ \textsc{Pure})
\end{aligned}$$

(d) Iterated and staged variant of control operations without eta-redexes.

Figure 3.6: Iterated and staged variant avoiding eta-redexes.

Assuming the target language has a primitive `if_then_else` of type $\underline{\mathsf{Bool}} \to \underline{\mathsf{A}} \to \underline{\mathsf{A}} \to \underline{\mathsf{A}}$, we define $\mathsf{ifthenelse}$ for statements with an arbitrary stack shape.

$$\begin{aligned}
&\mathsf{ifthenelse} \;:\; \underline{\mathsf{Bool}} \to \overline{\textsc{Stm}}\;\mathsf{RS}\;\mathsf{A} \to \overline{\textsc{Stm}}\;\mathsf{RS}\;\mathsf{A} \to \overline{\textsc{Stm}}\;\mathsf{RS}\;\mathsf{A}\\
&\mathsf{ifthenelse}_{(\mathsf{RS}\,=\,\bullet)}\;\mathsf{b}\;\mathsf{mt}\;\mathsf{me} \;=\; \underline{\mathbf{if}}\;\mathsf{b}\;\underline{\mathbf{then}}\;\mathsf{mt}\;\underline{\mathbf{else}}\;\mathsf{me}\\
&\mathsf{ifthenelse}_{(\mathsf{RS}\,=\,(\mathsf{Q}\;::\;\mathsf{QS}))}\;\mathsf{b}\;\mathsf{mt}\;\mathsf{me} \;=\; \lambda\mathsf{k} \Rightarrow \mathsf{ifthenelse}\;\mathsf{b}\;(\mathsf{mt}\;\mathsf{k})\;(\mathsf{me}\;\mathsf{k})
\end{aligned}$$

In case the stack shape is empty, we are already operating on values and can directly use the target language's `if_then_else`. In the other case, we abstract over the current context and recurse, passing the context to both the then-branch and the else-branch. This duplicates the static context and might result in a blow up of the size of the generated code. To avoid this duplication we can reify the context, give it a name and pass the reflected name to the two branches. Also, to generate the code in Figure 3.3 we performed a bit of partial evaluation (not shown here) to avoid generating an `if_then_else`, when the dynamic condition happens to be statically known to be True or False.

To implement defrec, we assume the target language has a `letrec` primitive of type $(\underline{(\mathsf{A} \to \mathsf{B})} \to \underline{\mathsf{A}} \to \underline{\mathsf{B}}) \to \underline{(\mathsf{A} \to \mathsf{B})}$.

$$\begin{aligned}
&\mathsf{defrec} \;:\; ((\underline{\mathsf{A}} \to \overline{\textsc{Stm}}\;\mathsf{RS}\;\mathsf{B}) \to \underline{\mathsf{A}} \to \overline{\textsc{Stm}}\;\mathsf{RS}\;\mathsf{B}) \to \underline{\mathsf{A}} \to \overline{\textsc{Stm}}\;\mathsf{RS}\;\mathsf{B}\\
&\mathsf{defrec}\;\mathsf{body} \;=\; \lambda\mathsf{a} \Rightarrow \textsc{Reflect}\;(\underline{\mathbf{letrec}}\;(\lambda\mathsf{f} \Rightarrow \lambda\mathsf{x} \Rightarrow\\
&\quad \textsc{Reify}\;(\mathsf{body}\;(\lambda\mathsf{y} \Rightarrow \textsc{Reflect}\;(\mathsf{f}\;\underline{@}\;\mathsf{y}))\;\mathsf{x}))\;\underline{@}\;\mathsf{a})
\end{aligned}$$

Granted, the implementation of defrec is rather complicated, but luckily we have the types to guide us with the insertion of staging annotations. The outermost call to REFLECT allows us to use defrec with statically known control flow. We necessarily need to reify the body in order to use it in $\underline{\mathbf{letrec}}$. Finally, the innermost REFLECT call reflects the recursive application to be used by body.

To summarize, in this section we have introduced a new type of contexts that allows us to statically distinguish between static and dynamic contexts. This helps us to avoid eta-redexes in the generated code. We have shown how to add primitives, $\mathsf{ifthenelse}$ and defrec to the source language, given that the target language has corresponding features.

3.7 Related Work

The amount of work on delimited continuations and on two-stage lambda calculus is vast, therefore we only compare the most closely related publications.

We base our basic control operator SHIFT0 and its CPS translation on prior work by Materzok and Biernacki [2011] who introduce a type system with subtyping, type inference and implicit coercions. In contrast, we require users to explicitly use LIFT. Additionally, they support answer type modification while we do not. Our statements are indexed by the stack shape, which is the list of answer types. Generally, to support answer type modification, effectful programs would have to be indexed by a tree of types.

We implement the CPS hierarchy first presented by Danvy and Filinski [1990]. However, our family of control operators is based on `shift0` instead of `shift`. They translate the entire program at a fixed level `n` of the CPS hierarchy, while we allow for different subterms of a program to live on different levels. While they provide types to ease understanding, their source language and meta language are untyped. We have types in the source language, use a typed meta language, and show how the two type systems cooperate.

This chapter is explicitly based on the classical work by Danvy and Filinski [1992]. The authors explain how to avoid administrative beta- and eta-redexes during a CPS transformation, by writing the translation itself in CPS. They also show the importance of distinguishing static from dynamic application and abstraction. In this chapter, we embrace many important insights from their work and extend them to iterated CPS.

3.8 Conclusion

In this chapter we have reimplemented and combined `shift0` [Materzok and Biernacki, 2011], abstracting control [Danvy and Filinski, 1990], and representing control [Danvy and Filinski, 1992] in a typed language combining different techniques and tradeoffs to ease the implementation. We are now able to write programs in a typed language using a family of control operators and generate code in any language that supports first-class functions. We extended our source language with primitives, `ifthenelse` and `defrec` and show how they interact nicely with our embedding. To get there, we reconstructed well known techniques and translated them to Idris and the typed setting. The individual steps were simple and guided by the types.

Prior work on the CPS hierarchy fixes a level of `n` control operators for the entire program upfront. Since we want to implement effect handlers, which are nested, we want to allow different parts of the program to live at different levels. It turns out that taking `shift0` rather than `shift` as the basis for the family of control operators is essential for this.

4 Evidence Passing

Abstract. In Chapter 2 we have seen effect handlers in explicit capability-passing style. The semantics of lexical effect handlers as well as their implementations use multi-prompt delimited control. They rely on freshly generated labels at runtime, which associate effect operations with their handlers.

In Chapter 3 we have revisited classical work on continuation-passing style (CPS). In iterated continuation-passing style functions receive not one but potentially multiple continuations. This allows us to translate nested delimiters and move between different levels of control.

The use of labels and multi-prompt delimited control as an implementation technique for lexical effect handlers is theoretically and practically unsatisfactory. In this chapter we show that typed lexical effect handlers do not need the full power of multi-prompt delimited control. We present a CPS translation for lexical effect handlers to pure System F in iterated CPS. It preserves well-typedness and simulates the traditional operational semantics. Importantly, it does so without any labels.

We introduce type-level regions and term-level subregion evidence. To find the correct delimiter, we interpret subregion evidence constructively. Whenever an effect operation is used, there must be evidence that a handler with the corresponding label is on the stack. We observe that this evidence does not merely tell us whether a label is on the stack, but exactly where it is.

Subregion evidence passing is the third ingredient of the compilation technique presented in this thesis.

In Chapter 5 we will evaluate our implementation technique for lexical effect handlers as a translation to STLC in iterated CPS. This translation uses monomorphisation and specialization of effect-polymorphic functions to achieve high performance of generated programs.

This chapter is based on the following publication: Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. "A Typed Continuation-Passing Translation for Lexical Effect Handlers". *Proc. ACM Conference on Programming Language Design and Implementation*, (PLDI) DOI: `https://doi.org/10.1145/3519939.3523710`

4.1 Programming with Regions and Evidence

In this section we informally introduce Λ_{Cap}, a language with lexical effect handlers, regions, and subregion evidence. We motivate the key ideas in this chapter by example. In Section 4.2 we formally define the syntax, typing, and operational semantics of Λ_{Cap}. In Section 4.3 we define a CPS translation from Λ_{Cap} to System F that preserves the operational semantics.

4.1.1 Using Lexical Effects

The purpose of an effect system is to guarantee *effect safety*. Intuitively effect safety means that every effect operation is eventually handled. More concretely, in the case of lexical effect handlers, it means that every capability is used in the extent of the corresponding handler. We call this dynamic extent a *region* [Tofte and Talpin, 1997, Grossman et al., 2002]. To guarantee effect safety, we keep track of the regions in which a computation can safely run.

As a starting point, consider the following simple example in System Ξ (Chapter 2) in explicit capability-passing style, which asks for two numbers and adds them:

```
effect Ask : Unit → Int

def askTwice(ask₁ : Ask, ask₂ : Ask) {
  do ask₁(()) + do ask₂(())
}
```

We explicitly abstract over and pass two different capabilities ask_1 and ask_2 of the same effect Ask. The function askTwice has the following type:

```
(Ask, Ask) → Int
```

It receives two capabilities of the effect Ask and returns an Int. In System Ξ all capabilities and blocks were second class. Naturally, this restriction ensures that all capabilities follow a stack discipline, since they cannot be returned. In this chapter we lift this restriction with an explicit region system.

Example 7. Consider the same function written in Λ_{Cap}. Each capability (*e.g.*, ask_1 and ask_2) is typed in their own type-level region (*e.g.*, r_1 and r_2).

```
def askTwice[r, r₁, r₂ ; n₁ : r ⊑ r₁, n₂ : r ⊑ r₂](ask₁ : Ask[r₁], ask₂ : Ask[r₂]) at r {
  do ask₁[n₁](()) + do ask₂[n₂](())
}
```

In Λ_{Cap} functions explicitly abstract over three things:

1. Functions abstract over *regions* (*e.g.*, r, r_1, and r_2). Inspired by the Single Effect Calculus [Fluet and Morrisett, 2004], in Λ_{Cap} there are no compound effects (such as *e.g.* $[\mathsf{r}_1, \mathsf{r}_2]$). Instead, a function like askTwice always only runs in a single region (*e.g.*, r).

2. Functions abstract over *subregion evidence* (*e.g.*, n_1 and n_2). We say that a region r *subsumes* another region r_1 (written $\mathsf{r} \sqsubseteq \mathsf{r}_1$) if all capabilities that can be used in r_1 can also be used in r. Subregion evidence witnesses this subsumption.

3. Functions abstract over *capabilities* (*e.g.*, ask_1 and ask_2). Every capability has a region where it is safe to use. When a capability like ask_1 is used, we require explicit evidence (*e.g.*, n_1) that the current region subsumes the region of the capability.

Consequently, the function askTwice has the following type:

$$\forall[\mathsf{r}, \mathsf{r}_1, \mathsf{r}_2 \,;\, \mathsf{r} \sqsubseteq \mathsf{r}_1, \mathsf{r} \sqsubseteq \mathsf{r}_2](\mathsf{Ask}[\mathsf{r}_1], \mathsf{Ask}[\mathsf{r}_2]) \to \mathsf{r}\, \mathsf{Int}$$

This type is very explicit about capabilities, regions and subregion evidence. It guarantees effect safety.

4.1.2 Handling Lexical Effects

Lexical effect handlers in explicit capability-passing style introduce a name for each capability. Consider the following example in System Ξ:

```
handle { ask$_1$ ⇒
  handle { ask$_2$ ⇒
    askTwice(ask$_1$, ask$_2$)
  } with { (u, k) ⇒ k(42) }
} with { (u, k) ⇒ k(43) }
```

In this example ask_1 and ask_2 are the names of two different capabilities for the Ask effect. We explicitly pass these capabilities to `askTwice`. Within `askTwice` there are two uses of the `ask` operation. Each of them will be handled by a different handler. The example evaluates to 85. It is possible to swap capabilities or to pass the same capability twice.

Example 8. Now consider the same program in Λ_{Cap}. Each handler introduces three things: firstly, a fresh region (*e.g.*, r_1) the handled program will run in. Secondly, evidence (*e.g.*, $\mathsf{n}_1 : \mathsf{r}_1 \sqsubseteq \top$) that the fresh region subsumes the outer one. Thirdly, a term-level capability (*e.g.*, $\mathsf{ask}_1 : \mathsf{Ask}[\mathsf{r}_1]$) containing the handler implementation. The capability's region is the freshly introduced region.

```
handle { [r$_1$ ; n$_1$ : r$_1$ ⊑ ⊤](ask$_1$ : Ask[r$_1$]) ⇒
  handle { [r$_2$ ; n$_2$ : r$_2$ ⊑ r$_1$](ask$_2$ : Ask[r$_2$]) ⇒
    askTwice[r$_2$, r$_1$, r$_2$ ; n$_1$, 𝟘](ask$_1$, ask$_2$)
  } with { (u, k) ⇒ k(42) }
} with { (u, k) ⇒ k(43) }
```

In Λ_{Cap}, each statement is checked in a region. In this example, the overall program is checked in region $\top$, the statement inside of the first handler is checked in region r_1, and the call to askTwice is checked in region r_2. The evidence each handler introduces witnesses that the freshly introduced region subsumes the outer region.

In comparison with System Ξ, the call to askTwice is more explicit. We explicitly apply functions to regions, evidence, and capabilities. Notably, the first evidence argument (*i.e.*, n_1) witnesses that r_2 subsumes r_1, and the second evidence argument (*i.e.*, $\mathbb{0}$) witnesses that r_2 subsumes r_2 itself, that is reflexivity. It is possible to swap capabilities or to pass the same capability twice, in which case the region- and evidence arguments must be adjusted accordingly.

4.1.3 Lexical Reasoning

Lexical handlers are not only useful to disambiguate different instances of the same effect [Bračevac et al., 2018a]. They also offer improved reasoning tools in the presence of higher-order functions [Zhang et al., 2016, Zhang and Myers, 2019]. Consider the following example, again in System Ξ:

```
handle { exc₁ ⇒
  def abort() { do exc₁(()) };
  handle { exc₂ ⇒
    abort()
  } with { (u, k) ⇒ "aborted two" }
} with { (u, k) ⇒ "aborted one" }
```

We install an exception handler and then define a function abort, which immediately fails. Because handlers in System Ξ are lexical, we know that the function abort will always abort to the handler which introduced the capability exc_1. This is the case, even if it is used under another exception handler and even if this handler was installed inside of another function. Hence the name "lexical" effect handler. As we have seen, operationally, abort closes over a fresh label that is bound to exc_1.

Example 9. Again, the same example in Λ_{Cap} is more explicit:

```
handle { [r₁, n₁ : r₁ ⊑ ⊤](exc₁ : Exc[r₁]) ⇒
  def abort[r, n : r ⊑ r₁]() at r { do exc₁[n](()) };
  handle { [r₂, n₂ : r₂ ⊑ r₁](exc₂ : Exc[r₂]) ⇒
    abort[r₂, n₂]()
  } with { (u, k) ⇒ "aborted two" }
} with { (u, k) ⇒ "aborted one" }
```

The function abort is region polymorphic. It abstracts over its region r. However, since it uses the capability `exc1` it is only safe to call abort in region r_1 or subregions of it. Therefore we have to constrain the region polymorphism and require that r subsumes r_1. Effectively, this makes sure that we only use abort in the dynamic extent of the handler that introduced exc_1. Concretely, to express constrained effect polymorphism, we abstract over evidence n that witnesses $r \sqsubseteq r_1$. We provide this evidence at the use of exc_1.

4.1.4 Operational Semantics and CPS Translation

Before going into the technical details of Λ_{Cap}, here we offer a high-level overview over the operational semantics of Λ_{Cap} and illustrate our CPS translation to System F.

Step One: Handler Passing

We present a typed CPS translation for lexical effect handlers to pure System F. To do so, it is necessary to get rid of runtime-generated labels. At first sight, this seems rather difficult, since labels are crucial to distinguish between different instances of the same effect at runtime and play an important role in the semantics of closures. After all, lexical reasoning is established by closing over labels.

The operational semantics of Λ_{Cap} is based on an abstract machine for multi-prompt delimited control. It generates fresh labels at runtime to disambiguate effect instances, and pushes frames with these labels onto a runtime stack to delimit the extent of effect operations. As mentioned in the introduction, we pass the handler implementation down to where it is used instead of allocating it on the runtime stack.

In Example 8, after taking a few steps and having handled the use of ask_1, the state of the machine is the following.

$$\begin{array}{l}\langle\ \mathbf{do\,cap}_{\texttt{@3a1}}\,\{\,(\mathsf{u},\,\mathsf{k})\Rightarrow\mathsf{k}(42)\,\}[\bullet](())\ \|\\ \quad 43+\square\ ::\ \#_{\texttt{@3a1}}\{\square\}\ ::\ \#_{\texttt{@b29}}\{\square\}\ ::\ \bullet\ \rangle\end{array}$$

It consists of a statement and a stack, separated by $\|$. Since we already executed the call to ask_1, the stack contains the frame $43 + \square$. It also contains two delimiters, one for the inner handler (marked with `@3a1`) and one for the outer handler (marked with `@b29`). The statement performs an effect and uses the capability $\mathbf{cap}_{\texttt{@3a1}}\{\,(\mathsf{u},\,\mathsf{k})\Rightarrow\mathsf{k}(42)\,\}$. The capability consists of the runtime label `@3a1` as well as the handler implementation.

Our CPS translation does not rely on runtime labels to find the correct handler implementation, because we pass the handler implementation down to its use-site (Chapter 2). The translation of this machine state is the following.

$$\begin{array}{l}((\lambda\mathsf{u}\Rightarrow\lambda\mathsf{k}\Rightarrow\mathsf{k}\,42)\,())\\ \quad(\lambda\mathsf{x}\Rightarrow(\lambda\mathsf{x}_1\Rightarrow\lambda\mathsf{k}_1\Rightarrow\mathsf{k}_1\,\mathsf{x}_1)\,(43+\mathsf{x}))\,(\lambda\mathsf{x}_2\Rightarrow\lambda\mathsf{k}_2\Rightarrow\mathsf{k}_2\,\mathsf{x}_2)\,\texttt{done}\end{array}$$

The first line shows the translation of the statement which uses the capability. We simply translate a capability by translating its handler implementation. The second line shows the translation of the stack. In this example the two delimiters partition the stack into three segments which we translate to three continuation arguments on the second line. The label of the capability is associated with the first delimiter. The handler implementation will correctly capture the first of the three continuations.

Step Two: Constructive Evidence

The operational semantics of Λ_{Cap} compares the labels at delimiters to capture the correct part of the stack when an effect operation is used. To guarantee that the search for a label is always successful, we let regions and subregion evidence be lists of labels. This is important for our proof of effect safety (Theorem 12), but neither regions nor evidence play any role computationally. But, as mentioned in the introduction, in our CPS translation we give *computational* meaning to evidence, and use evidence terms to find the correct handler. Surprisingly, this also works in the presence of higher-order functions and closures.

In Example 9, we used abort under a different handler for the same exception effect. Furthermore, we instantiated its region r with r_2 and passed evidence n_2 witnessing that $r_2 \sqsubseteq r_1$. After taking a few steps, the state of our abstract machine looks like this:

$$\langle\ \textbf{do}\ \textbf{cap}_{\texttt{@44c}}\ \{\ (u,\ k) \Rightarrow \texttt{"aborted one"}\ \}[\texttt{@8ab}\ ::\ \bullet\ ](())\ \|\ \\ \#_{\texttt{@8ab}}\ \{\ \Box\ \}\ ::\ \#_{\texttt{@44c}}\ \{\ \Box\ \}\ ::\ \bullet\ \rangle$$

In this example, the label @44c of the capability is associated with the second (that is, *outer*) delimiter on the stack. It is crucial that the continuation k is bound to the entire context up to this delimiter. Our abstract machine achieves this by comparing labels until the matching delimiter is found. How can we achieve the same in our CPS translation where no labels exist?

We observe that in this example the evidence @8ab :: • contains exactly the labels of the delimiters we have to skip. More generally, following [Xie et al., 2020], we will show that this is always the case (Theorem 13). The central idea is to take advantage of this fact and give computational content to subregion evidence in order to capture the correct part of the stack.

Concretely, we translate this machine state to the following term in System F:

$$(\lambda m \Rightarrow \lambda k \Rightarrow \lambda j \Rightarrow m\ (\lambda x \Rightarrow k\, x\, j))\ ((\lambda u \Rightarrow \lambda k \Rightarrow \lambda k_3 \Rightarrow k_3\ \texttt{"aborted one"})\ ()) \\ (\lambda x_1 \Rightarrow \lambda k_1 \Rightarrow k_1\, x_1)\ (\lambda x_2 \Rightarrow \lambda k_2 \Rightarrow k_2\, x_2)\ \texttt{done}$$

The first line corresponds to the statement that uses the capability. As before, the translated handler implementation is applied to the argument (). Importantly, we translate the singleton evidence @8ab :: • to the first term, called Lift (Chapter 3). Intuitively, it will capture the first continuation and push it onto the second one. This way, although there are no labels, the program executes correctly. In the next section, we start formalizing these ideas by introducing Λ_{Cap}.

4.2 The Language Λ_{Cap}

In this section, we formally introduce our source language Λ_{Cap}, a basic calculus with lexical effect handlers, regions, and subregion evidence. We define a type system and specify the operational semantics as an abstract machine.

The Λ_{Cap} calculus is sound and effect safe: we prove the usual theorems of Progress (Theorem 10) and Preservation (Theorem 11). Effect safety then follows as a corollary: whenever we use an effect, the corresponding handler is on the stack (Corollary 12).

Moreover, we establish the correspondence between type-level regions and term-level evidence (Corollary 13).

We have mechanized the formalization of Λ_{Cap} and its operational semantics in the Coq theorem prover [Bertot and Castéran, 2004], including the theorems of Progress and Preservation. The mechanization also includes the translation to System F as well as a proof of well-typedness preservation (Theorem 14).

4.2.1 Syntax

Figure 4.1 defines the syntax of Λ_{Cap}. We use fine-grain call-by-value [Levy et al., 2003] and syntactically distinguish between statements, which can have effects, and pure values.

Syntax of Statements Since statements can have effects, it makes for a clearer presentation to explicitly sequence them and to explicitly return values. Calling functions, performing effects, and handling effects are statements. We apply a function to a list of regions $\overline{\rho}$, a list of evidence terms $\overline{e}$, and a list of values $\overline{v}$. We use a capability to perform an effect with $\textbf{do}\, v_0[e](v)$, where v_0 is the capability, e is evidence, and v is the argument. We handle a statement with $\textbf{handle}\,\{\ldots\}\,\textbf{with}\,\{\ldots\}$. The handled statement receives a region r, evidence n, and a capability c. The handler receives an argument x and a continuation k.

Syntax of Values Functions (*i.e.*, $\{\,[\overline{r}\,;\,\overline{n : \gamma}](\overline{x : \tau})\,\textbf{at}\,\rho \Rightarrow s\}$) abstract over a list of type-level region parameters (*i.e.*, $\overline{r}$), a list of evidence variables (*i.e.*, $\overline{n : \gamma}$), and a list of term-level value parameters (*i.e.*, $\overline{x : \tau}$). Importantly, each function is defined to run exactly in a region ρ. The list of region parameters scopes over the evidence parameter types, the parameter types, the return type, the annotated region ρ, and the body s of the function. We omit type abstraction from this presentation since it is orthogonal to the rest of the calculus. Our mechanized formalization includes type abstraction and application.

Syntax of Evidence Evidence expressions are either an evidence variable n, the empty evidence $\mathbb{0}$ witnessing reflexivity of subregioning, or the composition of evidence $e \oplus e$, witnessing the transitivity of subregioning.

Syntax of Types Apart from the standard base and function types, Λ_{Cap} includes a type of capabilities. The type $\textbf{Cap}\,\rho\,\tau_1\,\tau_2$ indicates that a capability of this type can be used in a region ρ or any subregion and can be applied to an argument of type τ_1 to get a result of type τ_2.

Syntax of Regions and Constraints Regions ρ are either region variables r or the top-level region $\top$. Intuitively, the top-level region signals that no effect operations can be used. Constraints γ express subregion relationships.

Statements			
s	$::=$	$\textbf{val}\, x\ =\ s;\ s$	sequencing of statements
	$\mid$	$\textbf{return}\, v$	returning values
	$\mid$	$v[\overline{\rho}\,;\ \overline{e}](\overline{v})$	calling functions
	$\mid$	$\textbf{do}\, v[e](v)$	performing effects
	$\mid$	$\textbf{handle}\,\{\ [r\,;\ n](c) \Rightarrow s\ \}$	
		$\quad\textbf{with}\,\{\ (x,\ k) \Rightarrow s\ \}$	handling effects
Values			
v	$::=$	$x, f,\ c,\ \ldots$	variables
	$\mid$	$() \mid 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \ldots$	primitives
	$\mid$	$\{\ [\overline{r}\,;\ \overline{n\ :\ \gamma}](\overline{x\ :\ \tau})\ \textbf{at}\,\rho \Rightarrow s\}$	closures
Evidence			
e	$::=$	$n,\ \ldots$	evidence variables
	$\mid$	$\mathbb{0}$	reflexive evidence
	$\mid$	$e\ \oplus\ e$	transitive evidence
Types			
τ	$::=$	$\mathsf{Int} \mid \mathsf{Bool} \mid \ldots$	primitives
	$\mid$	$\forall[\overline{r}\,;\ \overline{\gamma}](\overline{\tau}) \to \rho\, \tau$	functions
	$\mid$	$\textbf{Cap}\,\rho\,\tau\,\tau$	capabilities
Regions			
ρ	$::=$	r	region variable
	$\mid$	$\top$	top-level region
Constraints			
γ	$::=$	$\rho\ \sqsubseteq\ \rho$	subregion

Γ	$::=$	$\emptyset$	empty environment
	$\mid$	$\Gamma,\ r$	region binding
	$\mid$	$\Gamma,\ n\ :\ \gamma$	evidence binding
	$\mid$	$\Gamma,\ x\ :\ \tau$	value binding

Figure 4.1: Syntax of Λ_{Cap}.

Statement Typing

$$\boxed{\underset{\uparrow}{\Gamma} \mid \underset{\uparrow}{\rho} \vdash \underset{\uparrow}{s} : \underset{\downarrow}{\tau}} \qquad \frac{\Gamma \mid \rho \vdash s_0 : \tau_0 \qquad \Gamma,\, x_0 : \tau_0 \mid \rho \vdash s : \tau}{\Gamma \mid \rho \vdash \textsf{val}\, x_0 = s_0;\ s : \tau}\ [\textsc{Val}] \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \mid \rho \vdash \textsf{return}\, v : \tau}\ [\textsc{Ret}]$$

$$\frac{\Gamma \vdash v_0 : \forall[\overline{r}\,;\,\overline{\gamma}](\overline{\tau}) \to \rho_0\, \tau_0 \qquad \overline{\Gamma \vdash e : \gamma[\overline{r \mapsto \rho}]} \qquad \overline{\Gamma \vdash v : \tau[\overline{r \mapsto \rho}]} \qquad \rho = \rho_0[\overline{r \mapsto \rho}]}{\Gamma \mid \rho \vdash v_0[\overline{\rho}\,;\,\overline{e}](\overline{v}) : \tau_0[\overline{r \mapsto \rho}]}\ [\textsc{App}]$$

$$\frac{\Gamma \vdash v_0 : \textsf{Cap}\, \rho'\, \tau_1\, \tau_2 \qquad \Gamma \vdash e : \rho \sqsubseteq \rho' \qquad \Gamma \vdash v : \tau_1}{\Gamma \mid \rho \vdash \textsf{do}\, v_0[e](v) : \tau_2}\ [\textsc{Do}]$$

$$\frac{\Gamma,\, r,\, n : r \sqsubseteq \rho,\, c : \textsf{Cap}\, r\, \tau_1\, \tau_2 \mid r \vdash s_0 : \tau \qquad \Gamma,\, x : \tau_1,\, k : \tau_2 \to \rho\, \tau \mid \rho \vdash s : \tau}{\Gamma \mid \rho \vdash \textsf{handle}\,\{\, [r\,;\, n](c) \Rightarrow s_0 \,\}\,\textsf{with}\,\{\, (x,\, k) \Rightarrow s \,\} : \tau}\ [\textsc{Handle}]$$

Value Typing

$$\boxed{\underset{\uparrow}{\Gamma} \vdash \underset{\uparrow}{v} : \underset{\downarrow}{\tau}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash 1 : \textsf{Int}}\ [\textsc{Lit}]$$

$$\frac{\Gamma,\, \overline{r},\, \overline{n : \gamma},\, \overline{x : \tau} \mid \rho \vdash s_0 : \tau_0}{\Gamma \vdash \{\, [\overline{r}\,;\, \overline{n : \gamma}](\overline{x : \tau})\ \textsf{at}\ \rho \Rightarrow s_0 \,\} : \forall[\overline{r}\,;\,\overline{\gamma}](\overline{\tau}) \to \rho\, \tau_0}\ [\textsc{Fun}]$$

Evidence Typing

$$\boxed{\underset{\uparrow}{\Gamma} \vdash \underset{\uparrow}{e} : \underset{\downarrow}{\gamma}} \qquad \frac{\Gamma(n) = \rho_1 \sqsubseteq \rho_2}{\Gamma \vdash n : \rho_1 \sqsubseteq \rho_2}\ [\textsc{EviVar}] \qquad \frac{}{\Gamma \vdash \mathbb{0} : \rho \sqsubseteq \rho}\ [\textsc{Reflexive}]$$

$$\frac{\Gamma \vdash e_1 : \rho \sqsubseteq \rho' \qquad \Gamma \vdash e_2 : \rho' \sqsubseteq \rho''}{\Gamma \vdash e_1 \oplus e_2 : \rho \sqsubseteq \rho''}\ [\textsc{Transitive}]$$

Figure 4.2: Type system of $\Lambda_{\textsf{Cap}}$.

4.2.2 Typing

Figure 4.2 defines the typing rules of $\Lambda_{\textsf{Cap}}$. We type statements, values, and evidence with different judgement forms. While all three are typed in an environment Γ containing region-, evidence-, and value bindings, only statements are typed in a given region ρ. Statements may perform effectful (that is, *serious* in the terminology of Reynolds [1972]) computation, which is only safe in certain contexts. In contrast, values are pure (that is, *trivial*) and can be used in any context.

Typing of Statements Rule Val types sequencing of statements. We type the two statements s_0 and s in the same region ρ of the compound statement. Returning a result from a computation (rule Ret) can be typed in any region. In rule App we apply a function v_0 to a list of regions $\overline{\rho}$, a list of evidence $\overline{e}$, and a list of arguments $\overline{v}$.

$$
\begin{array}{rcll}
l & ::= & \texttt{@a5f} \mid \texttt{@4b2} \mid \ldots & \\
v & ::= & \ldots \mid \textbf{cap}_l \,\{\, (x,\, k) \Rightarrow s \,\} \mid \textbf{resume}(\mathsf{H}) & \\
\mathsf{M} & ::= & \langle s \,\|\, \mathsf{K} \rangle & \text{executing} \\
 & \mid & \langle \textbf{do}\; v[w](v) \,\|\, \mathsf{K} \,\|\, \mathsf{H} \rangle & \text{unwinding} \\
 & \mid & \langle \textbf{resume}(\mathsf{H})(v) \,\|\, \mathsf{K} \rangle & \text{rewinding}
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{K} & ::= & \bullet \mid \mathsf{F} :: \mathsf{K} \\
\mathsf{H} & ::= & \bullet \mid \mathsf{F} :: \mathsf{H} \\
\mathsf{F} & ::= & \textbf{val}\, x = \Box;\, s \\
 & \mid & \#_l \,\{\, \Box \,\}
\end{array}
$$

Figure 4.3: Syntax of the abstract machine for Λ_{Cap}.

The type of v_0 is a function type annotated with a region ρ_0. The overall statement is typed in a region ρ. The premise $\rho = \rho_0[\overline{r \mapsto \rho}]$ requires that, after substituting the regions $\overline{\rho}$ for the region variables $\overline{r}$, both have to syntactically be the same. Note that we do not have any implicit subtyping here or elsewhere. Subregioning exclusively occurs through the passing of explicit subregion evidence. In rule Do, we type the use of a capability v_0 with evidence e and argument v. When a capability is used, we require explicit evidence that it is safe to do so. The evidence e witnesses that the region ρ of the statement subsumes the region of the capability ρ'. Again, there is no implicit subtyping and no subsumption rule. In rule Handle, the delimited statement s_0 is typed in a fresh region r. The evidence variable n witnesses that the fresh region r subsumes the outer region ρ of the whole statement. The capability c can be used in region ρ or any subregion. The statement s in the handler clause is typed in the same region as the overall statement. It receives a parameter x and a continuation k. The latter is a function which can only be called in precisely region ρ. This is because the continuation is itself effectful, and we want to ensure that calling it is safe.

Typing of Values The typing rules for variables Var and primitives Lit are standard. Rule Fun types functions. We type the body s_0 of the function in an environment extended with region parameters $\overline{r}$, evidence parameters $\overline{n : \gamma}$, and value parameters $\overline{x : \tau}$. Every function is annotated with a region ρ that specifies exactly the region that the function has to be called in. This region ρ is also the region in which we type the body s_0. The region parameters $\overline{r}$ may appear in the parameter types, the return type, the function's region ρ, and body s_0. This allows us to write region-polymorphic functions that can run in any region. As we have seen in Example 9, evidence parameters allow us to write region-polymorphic functions that are constrained to only run in a subregion of a given region.

Typing of Evidence Evidence variables are looked up in the typing environment. Reflexivity evidence $\mathbb{0}$ witnesses that every region is a subregion of itself, and transitivity evidence $e_1 \oplus e_2$ witnesses the transitivity of subregioning, which is reflected in their typing rules.

$$\begin{array}{lll}
\textit{(return)} & \langle \textbf{return}\, v \parallel \textbf{val}\, x \;=\; \square;\ s \;::\; \mathsf{K} \rangle & \to \\
 & \langle s[x \mapsto v] \parallel \mathsf{K} \rangle & \\
\textit{(push)} & \langle \textbf{val}\, x \;=\; s_0;\ s \parallel \mathsf{K} \rangle & \to \\
 & \langle s_0 \parallel \textbf{val}\, x \;=\; \square;\ s \;::\; \mathsf{K} \rangle & \\
\textit{(call)} & \langle \{\, [\overline{r}\,;\ \overline{n : \gamma}](\overline{x : \tau})\ \textbf{at}\ * \Rightarrow s_0 \,\}[\overline{*}\,;\ \overline{*}](\overline{v}) \parallel \mathsf{K} \rangle & \to \\
 & \langle s_0[\overline{r \mapsto *}, \overline{n \mapsto *}, \overline{x \mapsto v}] \parallel \mathsf{K} \rangle & \\
\textit{(handle)} & \langle \textbf{handle}\, \{\, [r\,;\ n](c) \Rightarrow s_0 \,\}\, \textbf{with}\, \{\, (x,\ k) \Rightarrow s \,\} \parallel \mathsf{K} \rangle & \to \\
 & \langle s_0[r \mapsto *, n \mapsto *, c \mapsto \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}] \parallel \#_l\, \{\, \square \,\} \;::\; \mathsf{K} \rangle & \\
 & \qquad \text{where } l \;=\; \texttt{generateFresh}() & \\
\textit{(pop)} & \langle \textbf{return}\, v \parallel \#_l\, \{\, \square \,\} \;::\; \mathsf{K} \rangle & \to \\
 & \langle \textbf{return}\, v \parallel \mathsf{K} \rangle & \\
\textit{(perform)} & \langle \textbf{do}\, \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}[*](v) \parallel \mathsf{K} \rangle & \to \\
 & \langle \textbf{do}\, \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}[*](v) \parallel \mathsf{K} \parallel \bullet \rangle & \\
\textit{(unwind)} & \langle \textbf{do}\, \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}[*](v) \parallel \textbf{val}\, x \;=\; \square;\ s \;::\; \mathsf{K} \parallel \mathsf{H} \rangle & \to \\
 & \langle \textbf{do}\, \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}[*](v) \parallel \mathsf{K} \parallel \textbf{val}\, x \;=\; \square;\ s \;::\; \mathsf{H} \rangle & \\
\textit{(forward)} & \langle \textbf{do}\, \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}[*](v) \parallel \#_{l'}\, \{\, \square \,\} \;::\; \mathsf{K} \parallel \mathsf{H} \rangle & \to \\
 & \langle \textbf{do}\, \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}[*](v) \parallel \mathsf{K} \parallel \#_{l'}\, \{\, \square \,\} \;::\; \mathsf{H} \rangle & \\
 & \qquad \text{where } l \neq l' & \\
\textit{(capture)} & \langle \textbf{do}\, \textbf{cap}_l\, \{\, (x,\ k) \Rightarrow s \,\}[*](v) \parallel \#_{l'}\, \{\, \square \,\} \;::\; \mathsf{K} \parallel \mathsf{H} \rangle & \to \\
 & \langle s[x \mapsto v,\ k \mapsto \textbf{resume}(\#_{l'}\, \{\, \square \,\} \;::\; \mathsf{H})] \parallel \mathsf{K} \rangle & \\
 & \qquad \text{where } l \;=\; l' & \\
\textit{(rewind)} & \langle \textbf{resume}(\mathsf{F}_1 \;::\; \mathsf{F}_2 \;::\; \mathsf{H})(v) \parallel \mathsf{K} \rangle & \to \\
 & \langle \textbf{resume}(\mathsf{F}_2 \;::\; \mathsf{H})(v) \parallel \mathsf{F}_1 \;::\; \mathsf{K} \rangle & \\
\textit{(resume1)} & \langle \textbf{resume}(\textbf{val}\, x \;=\; \square;\ s \;::\; \bullet)(v) \parallel \mathsf{K} \rangle & \to \\
 & \langle s[x \mapsto v] \parallel \mathsf{K} \rangle & \\
\textit{(resume2)} & \langle \textbf{resume}(\#_l\, \{\, \square \,\} \;::\; \bullet)(v) \parallel \mathsf{K} \rangle & \to \\
 & \langle \textbf{return}\, v \parallel \mathsf{K} \rangle &
\end{array}$$

Figure 4.4: Steps of the abstract machine for Λ_{Cap}.

4.2.3 Operational Semantics

Figure 4.3 lists the syntax and Figure 4.4 lists the reduction rules of the abstract machine semantics of Λ_{Cap}.

Labels As in Section 2.3.3, the semantics of Λ_{Cap} is given as a machine for multi-prompt delimited control. Our operational semantics uses labels l, which are freshly generated at runtime. Later in the CPS translation (Section 4.3), we will see how to avoid using runtime labels.

Runtime Values In order to specify the operational semantics, we extend the syntax of values with two additional runtime constructs. Firstly, runtime capabilities $\mathbf{cap}_l \, \{ \, (x, \, k) \Rightarrow s \, \}$ which consist of a label l and a handler implementation. Secondly, continuations $\mathbf{resume}(\mathsf{H})$ which contain a resumption H.

Machine States There are three different kinds of machine states. The component they all have in common is a runtime stack K which is a list of frames. A frame is either a sequencing frame $\mathbf{val}\, x \; = \; \square; \; s$ or a delimiter frame $\#_l \, \{ \, \square \, \}$ with a label l. The executing state has the form $\langle s \parallel \mathsf{K} \rangle$. It consists of the statement s under evaluation and the runtime stack K. The unwinding state consists of a performing statement, the runtime stack K, and a resumption H. In the unwinding state we unwind the stack K and push frames onto the resumption. The rewinding state consists of a resumption H, an argument v, and the runtime stack K. In the rewinding state we push frames from the resumption back onto the stack.

Reduction Rules The rules of the abstract machine in Figure 4.4 are mostly standard. While in Λ_{Cap}, we are very explicit about regions and evidence, we omit regions and evidence from this presentation of the operational semantics (*i.e.*, write $*$), because they are operationally irrelevant. But as we will see, they play an important role in our safety proof, in our CPS translation, and in our proof of simulation. The first rule *(return)* returns to the next frame on the stack. The *(push)* rule focuses on s_0 and pushes a frame on the stack. Rule *(call)* performs reduction by simultaneously substituting region arguments $\overline{\rho}$ for region variables $\overline{r}$, evidence arguments $\overline{e}$ for evidence variables $\overline{n}$, and values $\overline{v}$ for value parameters $\overline{x}$. Rule *(handle)* generates a fresh label and pushes a delimiter frame with this label onto the stack. The capability variable c is substituted by a capability that contains this label l and the handler implementation. Rule *(pop)* pops a delimiter off the stack upon normal return. Rule *(perform)* transitions from normal execution to unwinding. Rules *(unwind)* and *(forward)* move the next frame from the runtime stack onto the resumption. Rule *(capture)* executes the handler statement s with argument v. The continuation k is a resumption that rewinds the stack when called. This resumption must contain the delimiter frame $\#_{l'} \, \{ \, \square \, \}$. Rule *(rewind)* repushes the resumption onto the stack frame-by-frame until it is empty and rules *(resume1)* and *(resume2)* resume execution by returning the argument v to the stack.

$$\begin{array}{lllllll}
e & ::= & \ldots \mid w & \text{evidence value} & w & ::= & \bullet \mid l :: w & \text{evidence values} \\
\rho & ::= & \ldots \mid u & \text{runtime region} & u & ::= & \bullet \mid l :: u & \text{runtime regions}
\end{array}$$

$$\begin{array}{lll}
\mathcal{R}\llbracket \cdot \rrbracket & : & \mathsf{K} \rightarrow u \\
\mathcal{R}\llbracket \bullet \rrbracket & = & \bullet \\
\mathcal{R}\llbracket \textsf{val}\, x = \square;\, s :: \mathsf{K} \rrbracket & = & \mathcal{R}\llbracket \mathsf{K} \rrbracket \\
\mathcal{R}\llbracket \#_l \{ \square \} :: \mathsf{K} \rrbracket & = & l :: \mathcal{R}\llbracket \mathsf{K} \rrbracket
\end{array}$$

$$\begin{array}{lll}
\mathcal{N}\llbracket \cdot \rrbracket & : & e \rightarrow w \\
\mathcal{N}\llbracket \mathbb{0} \rrbracket & = & \bullet \\
\mathcal{N}\llbracket e_1 \oplus e_2 \rrbracket & = & \mathcal{N}\llbracket e_1 \rrbracket +\!\!+ \mathcal{N}\llbracket e_2 \rrbracket \\
\mathcal{N}\llbracket w \rrbracket & = & w
\end{array}$$

$$\frac{\emptyset \mid \mathcal{R}\llbracket \mathsf{K} \rrbracket \vdash s : \tau \qquad \vdash \mathsf{K} : \tau}{\vdash \langle s \parallel \mathsf{K} \rangle\ ok}\ [\textsc{Machine}]$$

$$\frac{u_0 = w +\!\!+ u_1}{\emptyset \vdash w : u_0 \sqsubseteq u_1}\ [\textsc{Evidence}]$$

$$\frac{\Gamma,\, x : \tau_1,\, k : \tau_2 \rightarrow u\, \tau \mid u \vdash s : \tau}{\emptyset \vdash \textsf{cap}_l \{ (x,\, k) \Rightarrow s \} : \textsf{Cap}\, (l :: u)\, \tau_1\, \tau_2}\ [\textsc{Capability}]$$

Figure 4.5: Proof invariants of the abstract machine.

4.2.4 Soundness

Λ_{Cap} satisfies the standard soundness properties.

Theorem 10 (Progress)**.**
If $\vdash\ M\, ok$, *then either* M *is of the form* $\langle \textbf{return}\, v \parallel \bullet \rangle$ *for some value* v, *or* $M \rightarrow M'$ *for some machine* M'.

Theorem 11 (Preservation)**.**
If $\vdash\ M\, ok$ *and* $M \rightarrow M'$ *then* $\vdash\ M'\, ok$.

In order to prove Progress and Preservation, we need to establish invariants, which are maintained by machine reduction. Figure 4.5 lists the most important concepts needed for our proofs.

Extended syntax We extend the syntax of values with evidence values w, and the syntax of regions with runtime regions u. Both are lists of labels. The top-level region $\top$ is the empty runtime region $\bullet$.

Connecting regions, evidence, and the stack To establish the connection between type-level regions ρ and the concrete runtime stack K, we define a semantic function

$\mathcal{R}[\![\cdot]\!]$ which computes the ordered list of labels of delimiters on the stack. This is the runtime region of a stack. We define a semantic function $\mathcal{N}[\![\cdot]\!]$, which normalizes evidence expressions to a list of labels.

Runtime typing In the typing of machine states, we type the statement s with the runtime region of the current stack K. In the typing of evidence values, we ensure that the evidence value w is the precise difference between runtime regions u_0 and u_1. In the typing of capabilities, the region of the capability is a runtime region where the label of the capability is the first element. In the handler implementation of the capability, the continuation is a function which has to run exactly in the rest of the runtime region of the capability.

Maintaining Invariants To maintain these invariants throughout reduction, in the full version of our reduction semantics, we substitute runtime regions for region variables and evidence values for evidence variables. For example, in step *(handle)* we substitute $l :: \mathcal{R}[\![\mathsf{K}]\!]$ for the region variable r and we substitute the singleton evidence value $l :: \bullet$ for the evidence variable n. In rule *(forward)* the evidence that occurs in the statement must be of the form $l' :: \mathcal{R}[\![\mathsf{K}]\!]$ before the step and just $\mathcal{R}[\![\mathsf{K}]\!]$ after the step.

4.2.5 Additional Properties

The runtime typing rules are designed to precisely reflect the invariants of the operational semantics. They very tightly constrain the possible machine states that can be encountered during execution. Effect safety follows as a corollary:

Corollary 12 (Effect Safety)**.**
If $\langle \textbf{do}\, \textbf{cap}_l \,\{\, (x,\, k) \Rightarrow s \,\}[e](v) \parallel \mathsf{K}\rangle$ *ok, then* l *is in* $\mathcal{R}[\![\mathsf{K}]\!]$.

Whenever we use a capability, a delimiter with the corresponding label is on the runtime stack.

However, we can prove an even stronger property. Whenever we use a capability, the evidence value precisely reflects the runtime stack. This corollary is inspired by the similarly named theorem of Xie et al. [2020].

Corollary 13 (Evidence Correspondence)**.**
If $\langle \textbf{do}\, v_0[e](v) \parallel \mathsf{K}\rangle$ *ok, then* $\mathcal{R}[\![\mathsf{K}]\!] = \mathcal{N}[\![e]\!] +\!\!+ (l :: u)$ *where* l *is the label of* v_0 *and* u *is some runtime region.*

This means that runtime evidence on the one hand and the labels in delimiters on the stack on the other hand are operationally redundant. The unwinding can *either* use evidence terms, *or* labels on the stack, since the two agree. Our proof uses both and establishes this fact. In the operational semantics we erase evidence terms as they do not have any significance at runtime. In the next section we are going to do the opposite: Erase labels in delimiter frames and purely rely on evidence terms to have the correct content at runtime.

4.3 Translation of Λ_{Cap} to System F

We now present the CPS translation of Λ_{Cap} to pure System F. Notably, in System F there are no labels. As a main result, by translating Λ_{Cap} into pure System F, we show that labels, recursive data types, or mutable state are not necessary to implement statically typed, lexical effect handlers.

We start from Λ_{Cap} in explicit capability-passing style (Chapter 2) without the restriction to second-class blocks, but with explicit regions and evidence. We translate into iterated continuation-passing style (Chapter 3). Every stack segment, delimited by a label, is represented by its own continuation argument. In Chapter 5 we evaluate our compilation technique and show that it enables compile-time optimizations for significant performance improvements. Since we target pure System F our compilation technique can target any language that supports first-class functions without any runtime support.

Figure 4.6 defines the translation of Λ_{Cap} to System F. A CPS translation is a translation of types and of terms. Our translation is defined over typing derivations of Λ_{Cap} (such as, $\mathcal{S}[\![\, \Gamma \mid \rho \vdash \; s \; : \; \tau \,]\!]$, abbreviated $\mathcal{S}[\![\, s \,]\!]$) to well-typed terms in System F.

Translation of Types

Base types, such as Int are left unchanged by the translation. Functions are translated to functions that abstract over a list of region variables at the type level, and over a list of evidence terms and a list of values at the term level. While evidence was computationally irrelevant in the operational semantics, it now plays a key role in finding the correct handler.

Interestingly, regions become answer types [Thielecke, 2003]. We use the familiar meta-definition $\textsc{Cps}\, R\, A$, defined as the type $(A \to R) \to R$ of computations in CPS with return type A and answer type R. Capabilities are translated to functions in CPS.

We translate region variables to type variables in System F and the top-level region to the empty type Void. Evidence terms are functions between effectful computations, as can be seen from the translation of evidence types. Since regions become answer types, region-polymorphic functions translate to answer-type polymorphic functions in CPS. Evidence terms adjust these answer types. They are constructive witnesses that we can move a computation from one region to a different one.

Translation of Terms

We translate variables, primitives, and functions in the obvious way. We translate evidence to functions that lift a computation to be compatible with a different region, *i.e.* answer type. The reflexivity evidence is translated to the polymorphic identity function, and transitivity of evidence amounts to function composition.

Return statements are translated to calls to the current continuation, and sequencing of statements is translated to push a frame onto the current continuation k, that is, the continuation first runs s and then continues with k. We translate function calls

$$
\begin{array}{lcl}
\mathcal{T}\llbracket\, \mathsf{Int} \,\rrbracket & = & \mathsf{Int} \\
\mathcal{T}\llbracket\, \forall[\overline{r}, \overline{\gamma}](\overline{\tau}) \to\!\rho\, \tau_0 \,\rrbracket & = & \overline{\forall r.}\ \overline{\mathcal{T}\llbracket c\rrbracket \to}\ \overline{\mathcal{T}\llbracket \tau\rrbracket \to}\, \textsc{Cps}\, \mathcal{T}\llbracket\rho\rrbracket\, \mathcal{T}\llbracket\tau_0\rrbracket \\
\mathcal{T}\llbracket\, \textbf{\textsf{Cap}}\, \rho\, \tau_1\, \tau_2 \rrbracket & = & \mathcal{T}\llbracket\tau_1\rrbracket \to \textsc{Cps}\, \mathcal{T}\llbracket\rho\rrbracket\, \mathcal{T}\llbracket\tau_2\rrbracket \\
\mathcal{T}\llbracket\, r \,\rrbracket & = & r \\
\mathcal{T}\llbracket\, \top \rrbracket & = & \mathsf{Void} \\
\mathcal{T}\llbracket\, \rho \,\sqsubseteq\, \rho' \,\rrbracket & = & \forall a.\ \textsc{Cps}\, \mathcal{T}\llbracket\rho'\rrbracket\, a \to \textsc{Cps}\, \mathcal{T}\llbracket\rho\rrbracket\, a \\
 & & \\
\mathcal{V}\llbracket\, x \,\rrbracket & = & x \\
\mathcal{V}\llbracket\, 1 \,\rrbracket & = & 1 \\
\mathcal{V}\llbracket\, \{\, [\overline{r}, \overline{n\ :\ \gamma}](\overline{x\ :\ \tau})\, \textbf{\textsf{at}}\, \rho \Rightarrow s \,\} \,\rrbracket & = & \overline{\Lambda r \Rightarrow}\, \overline{\lambda n \Rightarrow}\, \overline{\lambda x \Rightarrow}\, \mathcal{S}\llbracket s\rrbracket \\
 & & \\
\mathcal{E}\llbracket\, n \,\rrbracket & = & n \\
\mathcal{E}\llbracket\, \mathbb{0} \,\rrbracket & = & \Lambda a \Rightarrow \lambda m \Rightarrow m \\
\mathcal{E}\llbracket\, e_1 \,\oplus\, e_2 \,\rrbracket & = & \Lambda a \Rightarrow \lambda m \Rightarrow \mathcal{E}\llbracket e_1\rrbracket\, a\, (\mathcal{E}\llbracket e_2\rrbracket\, a\, m) \\
 & & \\
\mathcal{S}\llbracket\, \textbf{\textsf{return}}\, v \,\rrbracket & = & \lambda k \Rightarrow k\, (\mathcal{V}\llbracket v\rrbracket) \\
\mathcal{S}\llbracket\, \textbf{\textsf{val}}\, x \,=\, s_0;\ s \,\rrbracket & = & \lambda k \Rightarrow \mathcal{S}\llbracket s_0\rrbracket\, (\lambda x \Rightarrow \mathcal{S}\llbracket s\rrbracket\, k) \\
\mathcal{S}\llbracket\, v_0[\overline{\rho}, \overline{e}](\overline{v}) \,\rrbracket & = & \mathcal{V}\llbracket v_0\rrbracket \quad \overline{\mathcal{T}\llbracket\rho\rrbracket} \quad \overline{\mathcal{E}\llbracket e\rrbracket} \quad \overline{\mathcal{V}\llbracket v\rrbracket} \\
\mathcal{S}\llbracket\, \textbf{\textsf{do}}\, v_0[e](v) \,\rrbracket & = & \mathcal{E}\llbracket e\rrbracket \quad \mathcal{T}\llbracket\tau_2\rrbracket \quad (\mathcal{V}\llbracket v_0\rrbracket\, \mathcal{V}\llbracket v\rrbracket)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}\llbracket\, \textbf{\textsf{handle}}\, \{\, [r,\ n](c) \Rightarrow s_0 \,\}\, \textbf{\textsf{with}}\, \{\, (x,\ k) \Rightarrow s \,\} \,\rrbracket \ = \\
\qquad \textsc{Reset}\, ((\Lambda r \Rightarrow \lambda n \Rightarrow \lambda c \Rightarrow \mathcal{S}\llbracket s_0\rrbracket) \\
\qquad\qquad (\textsc{Cps}\, \mathcal{T}\llbracket\rho\rrbracket\, \mathcal{T}\llbracket\tau\rrbracket) \quad (\textsc{Lift}) \quad (\lambda x \Rightarrow \lambda k \Rightarrow \mathcal{S}\llbracket s\rrbracket))
\end{array}
$$

$$
\begin{array}{lcl}
\textsc{Cps}\, R\, A & = & (A \to R) \to R \\
\textsc{Reset} & : & \textsc{Cps}\, (\textsc{Cps}\, R\, A)\, A \to \textsc{Cps}\, R\, A \\
\textsc{Reset} & = & \lambda m \Rightarrow m\, (\lambda x \Rightarrow \lambda k \Rightarrow k\, x) \\
\textsc{Lift} & : & \forall a.\ \textsc{Cps}\, R\, a \to \textsc{Cps}\, (\textsc{Cps}\, R\, R')\, a \\
\textsc{Lift} & = & \Lambda a \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow \lambda j \Rightarrow m\, (\lambda x \Rightarrow k\, x\, j)
\end{array}
$$

Figure 4.6: Translation of Λ_{Cap} to System F.

to curried function application. The region arguments are type arguments, and the evidence- and value arguments are term arguments.

The two most complicated statements to translate are performing and handling. We translate the use of a capability v_0 with an argument v to an application of the capability to the argument. We then use the translated evidence e to adjust the resulting computation to run with the correct answer type. We translate handling statements to an application of the handled statement to three arguments: the answer type $\textsc{Cps}\,\mathcal{T}[\![\rho]\!]\,\mathcal{T}[\![\tau]\!]$, the singleton evidence $\textsc{Lift}$, and the capability $\lambda x \Rightarrow \lambda k \Rightarrow \mathcal{S}[\![s]\!]$. We use the meta function $\textsc{Reset}$ to apply the whole term to an empty continuation argument.

4.3.1 Typability Preservation

We translate well-typed programs in Λ_{Cap} to well-typed programs in System F.

Theorem 14 (Well-typedness of Translated Terms)**.**
If $\Gamma \mid \rho \vdash s : \tau$*, then* $\mathcal{T}[\![\,\Gamma\,]\!] \vdash \mathcal{S}[\![\,s\,]\!] : \textsc{Cps}\,\mathcal{T}[\![\,\rho\,]\!]\,\mathcal{T}[\![\,\tau\,]\!]$

Proof. Straightforward induction over the typing derivation. □

This theorem entails that, under the CPS translation, well-typed programs never get stuck, and that they always terminate. We mechanized the translation as well as the proof of Theorem 14 in the Coq proof assistant.

Example 15. To understand how regions in the source language and types in the target language are related, consider the following simple example where we install a handler and immediately use the capability it introduces.

$$\begin{array}{l} \textbf{handle}\,\{\,[\mathsf{r}\,;\,\mathsf{n}](\mathsf{exc}) \Rightarrow \textbf{do}\,\mathsf{exc}[\mathbb{0}](())\,\} \\ \quad \textbf{with}\,\{\,(\mathsf{x},\,\mathsf{k}) \Rightarrow \textbf{return}\,1\,\} \end{array}$$

This source statement is typed with $\emptyset \mid \top \vdash \ldots : \mathsf{Int}$, and we translate it to the following term in System F with overall type $\textsc{Cps}\,\mathsf{Void}\,\mathsf{Int}$.

$$\begin{array}{l} (\Lambda\mathsf{r} \Rightarrow \lambda\mathsf{n} \Rightarrow \lambda\mathsf{exc} \Rightarrow (\Lambda\mathsf{a} \Rightarrow \lambda\mathsf{m} \Rightarrow \mathsf{m})\,\mathsf{Int}\,(\mathsf{exc}\,())) \\ \quad (\textsc{Cps}\,\mathsf{Void}\,\mathsf{Int})\;\;(\textsc{Lift})\;\;(\lambda\mathsf{x} \Rightarrow \lambda\mathsf{k}_1 \Rightarrow \lambda\mathsf{k}_2 \Rightarrow \mathsf{k}_2\,1) \\ \quad (\lambda\mathsf{x} \Rightarrow \lambda\mathsf{k} \Rightarrow \mathsf{k}\,\mathsf{x}) \end{array}$$

The translation of the handled statement abstracts over a type and two terms. Maybe surprisingly, we apply it to *four* arguments. So how can this be welltyped? Recall that the handled statement (that is, $\textbf{do}\,\mathsf{exc}[\mathbb{0}](())$) is typed in region r and consequently is translated to a term in CPS of type $\textsc{Cps}\,\mathsf{r}\,\mathsf{Int}$. We instantiate the polymorphic answer type r with the type $\textsc{Cps}\,\mathsf{Void}\,\mathsf{Int}$, which results in the overall type $\textsc{Cps}\,(\textsc{Cps}\,\mathsf{Void}\,\mathsf{Int})\,\mathsf{Int}$ with two levels of control. This makes the application to the evidence, the capability, and the empty continuation type check. The capability has type $\mathsf{Unit} \to \textsc{Cps}\,(\textsc{Cps}\,\mathsf{Void}\,\mathsf{Int})\,\mathsf{Int}$. It discards the first continuation k_1 and returns to the second one k_2.

4.3.2 Simulation

In Section 4.2, we defined an operational semantics of Λ_{Cap}. In this section, we defined a CPS translation to System F. We now prove that the CPS translation simulates the operational semantics. To this end, we define a translation $\mathcal{M}[\![\ \cdot\ ]\!]$ of intermediate machine states M to well-typed terms in System F.

For each step the machine takes, there is a corresponding (possibly empty) sequence of steps between the translated terms.

Theorem 16 (Simulation)**.**
If $\mathsf{M} \to \mathsf{M}'$*, then* $\mathcal{M}[\![\ \mathsf{M}\ ]\!] \to^{*} \mathcal{M}[\![\ \mathsf{M}'\ ]\!]$.

Proof. By considering each case of the stepping relation. The *(perform)* step needs its own lemma, which we prove by induction on evidence terms. □

We translate statements to terms and stacks to evaluation contexts [Danvy, 2004]. We then define the translation of the machine in its executing state, which consists of a statement s and a stack K, as the plugging of the translation of the statement into the translation of the stack. The translation of the other machine states follows the same idea. We translate the empty stack to a special primitive function `done`, which will return the overall result of the program. It is called exactly once, when the machine is in its final state and we return to the empty stack.

The following corollary follows from Theorem 16. When we start from a closed, well-typed statement s and reduction of the machine results in a value v, then the translation of s applied to `done` evaluates to `done` applied to the translated result v.

Corollary 17 (Evaluation)**.**
If $\emptyset \mid \top \vdash\ s\ :\ \mathsf{Int}$ *and* $\langle\, s \parallel \bullet\, \rangle\ \to^{*}\ \langle\, \textbf{return}\, v \parallel \bullet\, \rangle$
then $\mathcal{S}[\![\ s\ ]\!]\ \ \mathtt{done}\ \ \to^{*}\ \ \mathtt{done}\ \ \mathcal{V}[\![\ v\ ]\!]$.

Although we do not have any labels generated at runtime, the CPS semantics exactly mimics the behavior of the operational semantics, which does have them.

We have not mechanized the proof of Theorem 16, because our mechanized translation introduces administrative reductions. Our mechanization includes a proof of the following weaker theorem ($\sim^{\beta\eta}$ denotes $\beta\eta$-equivalence):

Theorem 18 (Equivalence)**.**
If $\vdash\ \ \mathsf{M}\, ok$ *and* $\mathsf{M} \to \mathsf{M}'$*, then* $\mathcal{M}[\![\ \mathsf{M}\ ]\!]\ \sim^{\beta\eta}\ \mathcal{M}[\![\ \mathsf{M}'\ ]\!]$.

Corollary 17 still follows from this weaker theorem.

Example 19. Figure 4.7 lists a sequence of steps of the abstract machine and the corresponding sequence of steps of the translated machine states. In the example, we use a capability which is associated to an outer handler. It illustrates how the statement under reduction (highlighted in gray), the stack, and the resumption are translated, and how the translated term captures the correct number of continuations and reinstalls them.

The first steps are *(perform)*, *(forward)*, and *(capture)*. The evidence value `@3a1` :: $\bullet$ is precisely the offset we have to skip to get to the correct delimiter (Theorem 13).

$$(1)\ \langle\ \textbf{do cap}_{\texttt{@b29}}\{\ (\mathsf{u},\ \mathsf{k}_1) \Rightarrow \mathsf{k}_1(43)\ \}[\texttt{@3a1}\ ::\ \bullet](()) \ \|\ \#_{\texttt{@3a1}}\{\square\}\ ::\ \#_{\texttt{@b29}}\{\square\}\ ::\ \bullet\ \rangle \rightarrow$$

$$(2)\ \langle\ \textbf{do cap}_{\texttt{@b29}}\{\ (\mathsf{u},\ \mathsf{k}_1) \Rightarrow \mathsf{k}_1(43)\ \}[\texttt{@3a1}\ ::\ \bullet](()) \ \|\ \#_{\texttt{@3a1}}\{\square\}\ ::\ \#_{\texttt{@b29}}\{\square\}\ ::\ \bullet\ \|\ \bullet\ \rangle \rightarrow$$

$$(3)\ \langle\ \textbf{do cap}_{\texttt{@b29}}\{\ (\mathsf{u},\ \mathsf{k}_1) \Rightarrow \mathsf{k}_1(43)\ \}[\bullet](()) \ \|\ \#_{\texttt{@b29}}\{\square\}\ ::\ \bullet\ \|\ \#_{\texttt{@3a1}}\{\square\}\ ::\ \bullet\ \rangle \rightarrow$$

$$(4)\ \langle\ \textbf{resume}(\#_{\texttt{@b29}}\{\square\}\ ::\ \#_{\texttt{@3a1}}\{\square\}\ ::\ \bullet)(43)\ \|\ \bullet\ \rangle \rightarrow$$

$$(5)\ \langle\ \textbf{resume}(\#_{\texttt{@3a1}}\{\square\}\ ::\ \bullet)(43)\ \|\ \#_{\texttt{@b29}}\{\square\}\ ::\ \bullet\ \rangle \rightarrow$$

$$(6)\ \langle\ \textbf{return}\ 43\ \|\ \#_{\texttt{@b29}}\{\square\}\ ::\ \bullet\ \rangle \rightarrow$$

$$(7)\ \langle\ \textbf{return}\ 43\ \|\ \bullet\ \rangle$$

$$(1)\ \ (\textsc{Lift}\ ((\lambda \mathsf{u} \Rightarrow \lambda \mathsf{k}_1 \Rightarrow \lambda \mathsf{k}_2 \Rightarrow \mathsf{k}_1\ 43\ \mathsf{k}_2)\ ()))\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ \texttt{done} \rightarrow$$

$$(2)\ \ (\lambda \mathsf{k} \Rightarrow \lambda \mathsf{j} \Rightarrow (\lambda \mathsf{k}_1 \Rightarrow \lambda \mathsf{k}_2 \Rightarrow \mathsf{k}_1\ 43\ \mathsf{k}_2)\ (\lambda \mathsf{y} \Rightarrow \mathsf{k}\ \mathsf{y}\ \mathsf{j}))\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ \texttt{done} \rightarrow$$

$$(3)\ \ (\lambda \mathsf{k}_1 \Rightarrow \lambda \mathsf{k}_2 \Rightarrow \mathsf{k}_1\ 43\ \mathsf{k}_2)\ (\lambda \mathsf{y} \Rightarrow (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ \mathsf{y}\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x}))\ \texttt{done} \rightarrow$$

$$(4)\ \ (\lambda \mathsf{y} \Rightarrow (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ \mathsf{y}\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x}))\ 43\ \texttt{done} \rightarrow$$

$$(5)\ \ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ 43\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ \texttt{done} \rightarrow$$

$$(6)\ \ (\lambda \mathsf{k} \Rightarrow \mathsf{k}\ 43)\ (\lambda \mathsf{x} \Rightarrow \lambda \mathsf{k} \Rightarrow \mathsf{k}\ \mathsf{x})\ \texttt{done} \rightarrow$$

$$(7)\ \ (\lambda \mathsf{k} \Rightarrow \mathsf{k}\ 43)\ \texttt{done}$$

The statement under reduction is highlighted in *gray*, the first stack segment highlighted in *blue*, the second stack segment highlighted in *yellow*, and the bottom of the stack highlighted in *red*.

Figure 4.7: Example of step-by-step simulation.

$$\dfrac{\dfrac{\dots}{\emptyset \mid \texttt{@b29} :: \bullet \vdash \textbf{return}\ 43 : \mathsf{Int}}\,\textsc{Ret} \qquad \dfrac{\dfrac{}{\vdash \bullet : \mathsf{Int}}\,\textsc{Exit}}{\vdash \#_{\texttt{@b29}}\{\,\Box\,\} :: \bullet : \mathsf{Int}}\,\textsc{Handler}}{\vdash \langle\ \textbf{return}\ 43 \parallel \#_{\texttt{@b29}}\{\,\Box\,\} :: \bullet\ \rangle\ ok}\,\textsc{Machine}$$

$$\dfrac{\dfrac{\dfrac{\dots}{\vdash \lambda k \Rightarrow k\ 43 : (\mathsf{Int} \to \textsc{Cps}\ \mathsf{Void}\ \mathsf{Int}) \to \textsc{Cps}\ \mathsf{Void}\ \mathsf{Int}} \qquad \dfrac{\dots}{\vdash \lambda x \Rightarrow \lambda k \Rightarrow k\ x : \mathsf{Int} \to \textsc{Cps}\ \mathsf{Void}\ \mathsf{Int}}}{\vdash (\lambda k \Rightarrow k\ 43)\ (\lambda x \Rightarrow \lambda k \Rightarrow k\ x) : \textsc{Cps}\ \mathsf{Void}\ \mathsf{Int}} \qquad \dfrac{}{\dots}}{\vdash (\lambda k \Rightarrow k\ 43)\ (\lambda x \Rightarrow \lambda k \Rightarrow k\ x)\ \texttt{done} : \mathsf{Void}}$$

Figure 4.8: Typing derivations for state (6).

We translate it to the function LIFT which takes the first continuation and pushes it onto the second continuation. In state (3), after pushing the first delimiter onto the resumption, the stack consists of two segments. Therefore the translated term is applied to two continuations. The first continuation contains the translated resumption as a subterm. In step (5) we have pushed a delimiter back onto the stack and the translated term is again applied to two continuation arguments. This rewinding is achieved by the definition of LIFT and our translation of resumptions.

In Figure 19, we provide the typing derivations for machine state (6) and its translation. The machine state consists of a statement and a stack. The statement **return** 43 has type Int. It is typed in runtime region `@b29` :: $\bullet$, which is the runtime region $\mathcal{R}[\![\,\#_{\texttt{@b29}}\{\,\Box\,\} :: \bullet\,]\!]$ of the stack. The stack contains a delimiter with label `@b29` at type Int. Therefore the statement is translated to the term $\lambda k \Rightarrow k\,43$ at type CPS (CPS Void Int) Int. It is applied to two continuations. The first continuation $\lambda x \Rightarrow \lambda k \Rightarrow k\,x$ corresponds to the delimiter and will return the given value of type Int to the next continuation which expects a value of type Int. The second continuation `done` corresponds to the empty stack and will return the overall result once the program is done.

4.4 Related Work

We presented a CPS translation for lexical effect handlers. We first review and compare to existing work on lexical effect handlers, then we discuss related work on implementing dynamic effect handlers by CPS translation, on explicitly passing evidence, and on the relationship between regions and control effects.

4.4.1 Lexical Effect Handlers

There are a number of implementations of lexical effect handlers which do not guarantee effect safety. For example for OCaml [Kiselyov and Sivaramakrishnan, 2016], Scala [Brachthäuser and Schuster, 2017], and Java [Brachthäuser et al., 2018]. All of these use multi-prompt delimited control [Dybvig et al., 2007] under the hood, therefore their operational semantics is comparable to ours given in Section 4.2. However, because they do not have an effect system, they can generally express more programs. Of course this means that they can not guarantee effect safety and programs may crash because no delimiter with a corresponding label was found.

There are a number of languages with lexical effect handlers and an effect system which do guarantee effect safety [Zhang and Myers, 2019, Brachthäuser et al., 2020, Biernacki et al., 2019b]. All of these define effects as sets of effect instances a computation might use. In our type system in Section 4.2 we deviate from this presentation in two ways. Firstly, we avoid types depending on terms and introduce a fresh type-level region variable for each term-level capability. Secondly, like in the Single Effect Calculus [Fluet and Morrisett, 2004], we require the region of each function to be a single region variable. Operationally, all of these use some form multi-prompt delimited control, except for the open semantics, also presented by Biernacki et al. [2019b], which uses reduction under binders.

4.4.2 Dynamic Effect Handlers

Hillerström et al. [2017, 2020] present an implementation technique for dynamic effect handlers by CPS transformation. They present an abstract machine and a simulation result. In their abstract machine as well as in their CPS translated terms, each handler matches on tags at run time to decide whether it should handle an effect operation or forward it to an outer handler. In contrast, we present a CPS translation for lexical effect handlers in explicit capability-passing style. While our abstract machine searches for a matching label, our CPS translated terms do not. They produce untyped terms, where our CPS translation produces well-typed System F terms. They sketch how a typed and curried CPS translation might look like in Appendix B of [Hillerström et al., 2017], but do not present a fully worked out proof of well-typedness, which we do.

Leijen [2017b] presents a compilation technique for dynamic effect handlers by CPS translation. Their target language is untyped lambda calculus with a builtin handler construct which must be implemented by a runtime system on each target platform. In contrast, we present a translation for lexical effect handlers to well-typed System F, without any runtime system.

4.4.3 Explicit Evidence

Saleh et al. [2018] present a language with dynamic effect handlers where subeffect coercions are explicit. Their goal is to use the explicit information in these coercions to optimize effectful programs [Karachalias et al., 2021]. In a similar spirit, the language Λ_{Cap} that we present is very explicit about subregion evidence. While their coercions can be applied to arbitrary effectful expressions, we pass subregion evidence down to where effect operations are used and never coerce statements directly.

Xie et al. [2020, 2021] present an implementation of dynamic effect handlers where they explicitly pass evidence vectors. They translate programs to monadic style and use a monad for multi-prompt delimited control to capture resumptions. Their concept of evidence vectors is very different from the evidence that we have presented in this chapter. Their goal is for tail-resumptive handlers to not capture a resumption at all. Their source language features effect polymorphism based on effect rows [Leijen, 2014]. Guided by the effect types, they translate programs to explicitly pass evidence vectors, which are lists of pairs of labels and handler implementations. When an effect operation is used, they look up the corresponding handler in the evidence vector instead of on the stack. This is very similar to explicit capability-passing style. They prove that on a subset of programs their semantics correctly simulates dynamic handlers. The subset is characterized by *scoped resumptions*, intuitively that every resumption is invoked in the same context it was captured. They dynamically check this property, while we enforce the same property statically. Xie and Leijen [2021] lift the restriction of programs to only use scoped resumptions. Again, let us stress that their concept of evidence is very different from what we present in this chapter. Nevertheless, our formal treatment of the operational semantics is inspired by their concept of evidence correspondence.

4.4.4 Control Effects and Regions

Thielecke [2003] presents a language with `call/cc` and a type-and-effect system where effects are regions. He presents a CPS translation where regions become answer types and consequently region-polymorphic functions become answer-type polymorphic functions just as in our work. We expand upon his work by considering lexical effect handlers and delimited control instead of `call/cc`. The nesting of handlers makes it necessary to translate programs to iterated CPS. It is surprising that the same idea of regions as answer types also works in this setting.

Schuster et al. [2022] build upon the same idea and combine region-based resource management and lexical exception handlers in CPS. They guarantee cleanup of resources even when exceptions are thrown. While the problem they solve is different, their solution is the same as ours. They make subregion evidence explicit and endow it with computational content in their CPS translation. However, in comparison to exception handlers, effect handlers are more general because they can resume computation. Consequently, our formalization is more challenging because we have to reify resumptions as values. We leave combining their approach to resource management with lexical effect handlers to future work.

4.5 Conclusion

In this chapter we have presented a CPS translation from Λ_{Cap} to well-typed System F with neither labels nor runtime constructs. It simulates the standard operational semantics which does use labels and special runtime constructs. It works in the presence of constrained effect-polymorphism and first-class functions.

There is room for improvement. The effect operations supported by Λ_{Cap} must have monomorphic types. In the future, it would be interesting to extend the approach to more advanced forms. While we believe that type-polymorphic effect operations are a straightforward extension, effect-polymorphic effect operations or bidirectional effects [Zhang et al., 2020] could hold some interesting challenges.

Moreover, we statically enforce the property of scoped resumptions [Xie et al., 2020], which means that resumptions must be called in exactly the same region they were based in. This rules out programs where an effect operation dynamically changes how operations in the rest of the program are to be handled. A useful example would be the interleaving of two push streams. While we believe that programs like these go against the spirit of lexical effect handlers, in the future we would like to investigate how to support these use cases as well.

5 Compiling Handlers

Abstract. In Chapter 2 we have seen that effect handlers encourage programmers to abstract over repeated patterns of complex control flow. This abstraction, however, comes at a significant price in performance. Moreover, we have seen a translation to explicit capability-passing style, and a semantics with multi-prompt delimited control which requires special runtime support.

In Chapter 3, effectful programs are indexed by their stack shape, which is the list of answer types from innermost to outermost. Guided by these stack shapes, we translate programs to iterated continuation-passing style, making the flow of continuations explicit. We have used a two-level lambda calculus to avoid generating administrative beta redexes, and we have explained how to avoid generating administrative eta redexes.

In Chapter 4, starting from a language in explicit capability-passing style, we have seen an operational semantics with multi-prompt delimited control, a translation to iterated continuation-passing style, and a proof of simulation between the two. The source language supports constrained effect polymorphism with explicit subregion evidence. Our translation can target any language with first-class functions, without any special runtime support.

In this chapter, we finally put everything together and evaluate the performance of our compilation technique both theoretically and practically. There are two different aspects to the performance of abstractions like effect handlers: enabling compile time optimization and optimizing the language runtime. In this chapter, we are only concerned with the former. We show that, under some conditions, we are able to fully eliminate the abstraction overhead of effect handlers at compile time. To evaluate the practical performance, we present benchmarks where we compare generated code with Koka, Multicore OCaml, and Chez Scheme. The benchmarks indicate that our translation offers significant speedups for programs which heavily use effect handlers and shows competitive performance for examples with only simple uses of effect handlers.

Both, our theoretical and practical results crucially rely on the combination of explicit capability-passing style, iterated continuation-passing style, and explicit subregion evidence. By making all this information explicit, it is possible to aggressively specialize effectful programs to their context, simply by monomorphizing effectful functions, inlining capabilities, continuations, and evidence, and reducing the resulting program.

This chapter is based on the following publication: Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. "Compiling Effect Handlers in Capability-Passing Style." Proc. ACM Program. Lang., 4 (ICFP), DOI: https://doi.org/10.1145/3408975

5.1 Capabilities, Continuations, and Evidence

In this section we describe how we use the compilation technique presented in this thesis to generate high-performance code. In previous chapters we have made more and more information explicit. Now we use this explicit information to specialize effectful programs to their context.

Figure 5.1 shows an example program, again adapted from Danvy and Filinski [1990] and written in λ_{Cap}, which will introduce in this chapter. The effectful function choice uses two effects Fork and Fail. The signatures of these effect operations are given by the following two global signatures.

$$\textbf{effect}\,\mathsf{Fork} : \mathsf{Unit} \to \mathsf{Bool} \qquad\qquad \textbf{effect}\,\mathsf{Fail} : \mathsf{Unit} \to \mathsf{Void}$$

The function choice chooses a number between the given argument n and 1. If n is smaller than 1, it fails. Otherwise it forks the computation to decide if it immediately returns n or recursively calls itself with a decremented argument.

Handling effects The meaning of the effects Fork and Fail in choice is left open. It is written in explicit capability-passing style (Chapter 2). It abstracts over capabilities fork and fail and uses them to perform the corresponding effects. The function handledChoice handles Fork and Fail. It gathers all results into a list. The handler for Fork calls the continuation k twice, once with True and once with False. It expects the results of these two calls to be lists, appends them, and answers with the appended list. The implementation for fail ignores k and immediately answers with the empty list. The two handlers introduce capabilities which we explicitly pass to the effectful function choice.

Effect safety To guarantee effect safety and to guide our translation we index effectful computations with their stack shape (Chapter 3). In our example, both handlers have the same answer type IntList and the stack shape at the call-site of choice is thus [IntList, IntList]. To achieve answer-type safety (*i.e.*, capturing and applying the continuation is type safe) and effect safety (*i.e.*, all effects are eventually handled), the type system of λ_{Cap} indexes the types of capabilities and the types of effectful functions by the stack shape they assume. We write the type of the choice function at this call-site as:

$$\overbrace{\mathsf{Fork}[\mathsf{IntList},\mathsf{IntList}] \to \mathsf{Fail}[\mathsf{IntList},\mathsf{IntList}]}^{\text{capabilities}} \to \overbrace{\mathsf{Int} \to [\mathsf{IntList},\mathsf{IntList}] \quad \mathsf{Int}}^{\text{effectful function type}}$$

The function choice is an effectful function that takes a capability for Fork, a capability for Fail, and an Int, and returns an Int. It assumes a stack shape IntList, IntList. To safely invoke an effect operation, the stack shape of the computation at the invocation site and the stack shape of the capability have to agree.

```
def choice[fork : Fork, fail : Fail](n) {
  if (n < 1) do fail()
  else if (do fork())
    return n
  else choice(n − 1)
}
```

```
def handledChoice(n) {
  handle { fork ⇒
    handle { fail ⇒
      Con(choice[lift fork, fail](n), Nil)
    } with { (u, k) ⇒ Nil }
  } with { (u, k) ⇒
    append(do k(True), do k(False)) }
}
```

(a) Source program written in λ_{Cap} in capability-passing style.

```
let choice = λfork ⇒ λfail ⇒
  letrec loop = λn ⇒ λk1 ⇒ λk2 ⇒
    if (n < 1) then fail () k1 k2
    else fork () (λx ⇒ λk3 ⇒
      if x then k1 n k3 else loop (n − 1) k1 k3)
      k2
  in loop
```

```
let handledChoice = λn ⇒
  let fork = λu ⇒ λk ⇒
        append (k True) (k False) in
  let fail = λu ⇒ λk1 ⇒ λk2 ⇒ k2 Nil in
  let liftedFork = λu ⇒ λk1 ⇒ λk2 ⇒
        fork () (λx ⇒ k1 x k2) in
  choice liftedFork fail n
      (λx1 ⇒ λk2 ⇒ k2 (Con x1 Nil))
      (λx2 ⇒ x2)
```

(b) Code generated from λ_{Cap} in iterated CPS.

```
letrec choiceForkFail =
  λn ⇒ λk1 ⇒ λk2 ⇒
    if (n < 1) then k2 Nil
    else
      let x1 = k1 n k2 in
      let x2 = choiceForkFail (n − 1) k1 k2 in
        append x1 x2
```

```
let handledChoice = λn ⇒
  choiceForkFail n
   (λx1 ⇒ λk2 ⇒ k2 (Con x1 Nil))
   (λx2 ⇒ x2)
```

(c) Code generated from $\mathbb{\lambda}_{\mathsf{Cap}}$ with inlined handlers (highlighted in gray).

Figure 5.1: Running example in our language λ_{Cap} and its translation into CPS.

Lifting capabilities Each handler introduces a fresh region and the regions of nested handlers are related by subregion evidence (Chapter 4). Here we view regions as stack shapes. To exploit the information we get from the static context of effectful programs, we implement region polymorphism by monomorphization[2], a common technique in performance-oriented compilers [Stroustrup, 1997, Alexandrescu, 2010, Anderson et al., 2016]. This specializes effectful programs to their stack shape. Since we created the capability fork at the outer handler, its type is Fork[IntList]. To use it inside of the inner handler, as an argument to choice, we have to explicitly adapt it with **lift**. This way, the capability can be used in a context with the larger stack shape [IntList, IntList].

Compilation of λ_{Cap} We translate our source language λ_{Cap} to STLC (with **letrec**). Directed by the statically known stack shape, our translation introduces one continuation argument for every delimiting handler. Figure 5.1 shows the result of specializing choice to the stack shape [IntList, IntList] at its call-site. The generated code uses two continuations corresponding to the two delimiters for Fork and Fail. At its call-site we supply five arguments to choice: two capabilities, the argument n, and two continuations corresponding to the two delimiters. The first continuation represents the context around choice at its call-site. It is itself in CPS and takes a continuation. The second continuation is the empty continuation. The capability fork has been translated at stack shape [IntList] and so abstracts over only one continuation. Here we see that **lift** has computational content as it adjusts fork to be compatible to a context with two continuations by composing them.

If we were using the same choice function at a different call-site, within for example three enclosing handlers, it would be typed at a different stack shape and consequently specialized differently:

```
let choice = $\lambda$fork ⇒ $\lambda$fail ⇒
  letrec loop = $\lambda$n ⇒ $\lambda k_1$ ⇒ $\lambda k_2$ ⇒ $\lambda k_3$ ⇒
    if (n < 1) then fail () $k_1$ $k_2$ $k_3$
    else fork () ($\lambda$x ⇒ $\lambda k_4$ ⇒ $\lambda k_5$ ⇒
      if x then $k_1$ n $k_4$ $k_5$ else loop (n − 1) $k_1$ $k_4$ $k_5$ ) $k_2$ $k_3$
  in loop
```

We abstract over one more continuation and apply functions to one more continuation (highlighted in gray). Through monomorphization we have created a second specialized version of the same function. Operationally, only the number of elements in the stack shape, *i.e.* the number of continuations, matters. In a typed setting, though, we have to specialize to the types contained in the stack shape. While this translation specializes choice to different stack shapes, it still abstracts over capabilities fork and fail.

[2]http://mlton.org/Monomorphise

Compilation of $\boldsymbol{\lambda}_{\mathsf{Cap}}$ Our running example does not treat capabilities as first class [Osvald et al., 2016] and so can be typed under the more restrictive rules of $\boldsymbol{\lambda}_{\mathsf{Cap}}$, which enforce a second-class usage of capabilities. Consequently, we can use our second translation, which works for $\boldsymbol{\lambda}_{\mathsf{Cap}}$ only, to also specialize the code to the concrete handler implementations. In this translation we distinguish between static and dynamic abstractions and reduce capability abstractions and applications statically. This way, the implementations of Fork and Fail provided by the corresponding handlers are inlined into the body of choice. Figure 5.1 shows the final code we generate for this example. We chose the name choiceForkFail to reflect the specialization to these handler implementations. The cost of the handler abstraction has been fully removed and the function is specialized to both the effect operation implementations and the stack shape at its call-site.

There is no runtime search for a matching handler like in Koka [Leijen, 2017b], Eff [Plotkin and Pretnar, 2013], Frank [Lindley et al., 2017], Multicore OCaml [Dolan et al., 2014], or Helium [Biernacki et al., 2019b]. Instead, the implementations of the effect operations have been inlined into the body of choice. Correspondingly, the call-site in handledChoice does not provide capabilities anymore, it only delimits the control effects. Furthermore, there is no search for a delimiter with a matching prompt on the stack, like in the operational semantics of for example Scala Effekt [Brachthäuser and Schuster, 2017], Java Effekt [Brachthäuser et al., 2018], Olaf [Zhang and Myers, 2019], or newer versions of Koka [Xie et al., 2020, Xie and Leijen, 2021]. We directly invoke the corresponding continuation.

To sum up our approach: We start with a program in explicit capability-passing style. Effectful functions and capabilities are indexed by the stack shape. Capabilities need to be explicitly lifted to adjust them to the stack shape. Using this information, we specialize functions written in λ_{Cap} to work with the correct number of continuations. For a refined sub-language $\boldsymbol{\lambda}_{\mathsf{Cap}}$, we guarantee that capabilities are always inlined. We specialize functions to their context and remove the cost associated with handler abstractions. In the following sections we will formally develop these ideas.

5.2 The Language λ_{Cap}

In this section we formally introduce λ_{Cap}, a language in explicit capability-passing style (Chapter 2), where effectful programs are indexed by stack shapes (Chapter 3), and where lifting of capabilities is explicit (Chapter 4).

5.2.1 Syntax

Figure 5.2 defines the syntax of λ_{Cap}. Like other presentations of languages with effect handlers [Pretnar, 2015, Kammar and Pretnar, 2017, Hillerström et al., 2017], our language is based on a fine-grain call-by-value lambda calculus [Levy et al., 2003]. That is, we syntactically distinguish between statements and expressions. Only statements can have effects. We also syntactically distinguish between expressions and capabilities.

Statements

s	$::=$	$\textsf{\textbf{val}}\, x \;=\; s;\, s$	sequencing
	$\mid$	$\textsf{\textbf{return}}\; e$	returning
	$\mid$	$e(e)$	calling
	$\mid$	$\textsf{\textbf{do}}\, h(e)$	performing
	$\mid$	$\textsf{\textbf{handle}}\; \{\; c \Rightarrow s \;\}\; \textsf{\textbf{with}}\; h$	handling

Expressions

e	$::=$	$\textsf{True} \mid \textsf{False} \mid \ldots$	primitive constants
	$\mid$	x	term variables
	$\mid$	$(x \,:\, \tau) \Rightarrow s$	lambda abstraction
	$\mid$	$\textsf{\textbf{fix}}\, f\; (x \;:\; \tau) \Rightarrow s$	recursive abstraction
	$\mid$	$[c \,:\; F[\xi]\,] \Rightarrow e$	capability abstraction
	$\mid$	$e[h]$	capability application

Capabilities

h	$::=$	$c \mid k$	capability variables
	$\mid$	$\{\, (x,\, k) \Rightarrow s\, \}$	handler implementation
	$\mid$	$\textsf{\textbf{lift}}\, h$	lifted capability

Types

τ	$::=$	$\textsf{Int} \mid \textsf{Bool} \mid \ldots$	base types
	$\mid$	$\tau \to\!\xi\; \tau$	effectful function type
	$\mid$	$F[\xi] \to \tau$	capability function type

Operation Names

$F \quad ::= \quad \textsf{Fork} \mid \textsf{Fail} \mid \textsf{Emit} \mid \textsf{Resume}_i \mid \ldots$

Operation Signatures

$\Sigma \quad ::= \quad \emptyset \mid \Sigma,\, F \;:\; \tau \to \tau'$

Type Environment

$\Gamma \quad ::= \quad \emptyset \mid \Gamma,\, x \;:\; \tau$

Capability Environment

$\Theta \quad ::= \quad \emptyset \mid \Theta,\, c \;:\; F[\xi]$

Stack Shape

$\xi \quad ::= \quad \bullet \mid \tau \,::\, \xi$

Figure 5.2: Syntax of $\lambda_{\textsf{Cap}}$.

Statements Both, calling functions (*i.e.*, $e(e')$) and using capabilities (*i.e.*, $\textbf{do}\, h(e)$) are considered effectful. The latter performs the effect of the capability. Expressions are embedded into statements with $\textbf{return}\, e$ and we use the syntax $\textbf{val}\, x = s_0;\ s$ to sequence the evaluation of the two statements s_0 and s. The result of s_0 is available in s under the name x. Finally, we handle effectful programs with $\textbf{handle}\,\{\, c \Rightarrow s \,\}\,\textbf{with}\, h$. The capability variable c will be bound to the handler implementation h in the statement s. The handler also installs a delimiter for the continuation which is captured when c is used.

Expressions As usual, the syntax of expressions includes primitive constants (like 5, True, and Nil), function abstraction (*i.e.*, $(x : \tau) \Rightarrow s$), and recursive function abstraction (*i.e.* $\textbf{fix}\, f\ (x : \tau) \Rightarrow s$). Additionally, capability abstraction (*i.e.*, $[c : F[\xi]\,] \Rightarrow e$) binds a capability c for effect operation F, which is usable in the expression e in a context with stack shape ξ. Calling an effectful function with an argument can have control effects and thus is not an expression but a statement. In contrast, capability application (*i.e.*, $e[h]$) is pure and results in an expression. Similarly, primitive operators (like $\mathsf{append}(e,\ e)$) cannot have control effects and are trivial expressions.

Capabilities We separate expression variables from capability variables. The latter are drawn from a different namespace (*e.g.*, fork, fail, or k). Similarly, we use the metavariables x for term variables and c and k for capability variables. This stratification into expressions and capabilities is not strictly necessary in λ_{Cap}, but it will become important in $\boldsymbol{\lambda}_{\mathsf{Cap}}$. To facilitate comparison we use the same syntax of terms for both languages. The $\textbf{lift}\, h$ construct adjusts a capability h to be compatible with a larger stack shape. Capabilities are handler implementations constructed with $\{\, (x,\ k) \Rightarrow s \,\}$. The argument x and the continuation k are bound in the implementation of the effect operation given by s. As we will see, we model continuations as capabilities and thus k has to be invoked with $\textbf{do}\, k(e)$.

5.2.2 Typing

Types include base types (*e.g.*, Int and Bool), effectful function types $\tau_1 \rightarrow^{\xi} \tau_0$, and capability abstractions $[\, F[\xi]\,] \rightarrow \tau$. Effectful function types should be read as follows. Given τ_1, the function can only be called in a context with stack shape ξ to produce a result of type τ_0. Stack shapes are comma separated lists of types τ, representing the types at the delimiters (*i.e.* handlers) from innermost to outermost. They serve a similar purpose like effect rows of Koka [Leijen, 2017b] or Links [Hillerström and Lindley, 2016]. Like effect rows in Koka and Links, our stack shapes guarantee that our control effects are handled and all continuations are correctly delimited. However, unlike effect rows, stack shapes are *ordered*. As an example, the stack shape [Int, String] describes a context with an inner handler at type Int and an outer handler at type String.

Capability abstractions take a capability parameter. Like effectful functions, the type of each capability parameter $F[\xi]$ is restricted to a specific stack shape ξ. We use the meta-variable F to denote a globally fixed set of operation names and assume a global signature environment Σ that maps operation names to their input and output types. We model continuations as capabilities and include a family Resume_i in the set of operation names. Each syntactic occurrence of **handle** ... **with** ... induces a distinct operation name Resume_i. The typing of the corresponding **handle** statement fully determines the signature of Resume_i in Σ.

Following the distinction between expressions and capabilities, we also assume two separate environments. A type environment Γ that assigns variables x to types τ and a capability environment Θ that associates capability variables c with operation names F and stack shapes ξ. This separation, again, is not necessary in λ_{Cap} but will be in $\lambda\!\!\!\lambda_{\mathsf{Cap}}$.

Typing Rules

The typing rules in Figure 5.3 are defined by three mutually recursive typing judgements – one for each syntactic category. The judgement form $\Theta \mid \Gamma \mid \xi \vdash s : \tau$ assigns a type τ to the statement s. Moreover, the statement s is checked in a stack shape ξ.

The typing rules include standard rules for variables (Var), abstractions (Lam), recursive abstractions (Fix), and applications (App). Sequencing with rule Val requires that the stack shapes of the two statements s_0 and s agree. Similarly in rule Do the stack shape of the used capability and the **do** statement have to agree. In rule Ret, the resulting statement is compatible with any stack shape ξ.

The rules for capability abstraction (CapLam) and application (CapApp) are similar to the corresponding rules for value abstraction and application. However, capability abstraction introduces the capability variable c in the capability environment Θ and capability application uses the capability typing judgement $\Theta \mid \Gamma \vdash h : F[\xi]$ to check h against operation name F in stack shape ξ.

The three most interesting rules are Handle, CapLift, and CapHandler. They require some detailed explanation. Handlers introduce delimiters for the continuations captured by the effect operation they handle. This becomes visible in rule Handle. While in the conclusion, we type a statement **handle** $\{ c \Rightarrow s \}$ **with** h against a stack shape ξ, the premises can assume a larger stack shape $\tau :: \xi$. By installing a delimiter, the statement s can safely use the capability c, which has additional control effects at answer type τ. To guarantee answer type safety, the return type τ of the delimited statement s and the innermost answer type of the larger stack shape $\tau :: \xi$ have to agree.

Our type system does not support implicit effect subtyping. Instead, capabilities need to be lifted explicitly. Take the following ill-typed example:

handle $\{ c_1 \Rightarrow$ **handle** $\{ c_2 \Rightarrow$ **do** $c_1(x) \}$ **with** $h_2 \}$ **with** h_1

We bind a capability variable c_1 at an outer handler, but want to use it inside of a nested inner handler. While using the capability within the inner handler would be safe, the stack shapes do not match up. To account for this, we allow explicit lifting of capabilities with **lift** h. In the example, we could thus invoke **do** (**lift** c_1)(x). Rule

Expression Typing

$$\boxed{\Theta \mid \Gamma \vdash e : \tau} \qquad \frac{\Gamma(x) = \tau}{\Theta \mid \Gamma \vdash x : \tau}\,[\textsc{Var}]$$

$$\frac{\Theta \mid \Gamma, x : \tau_1 \vdash s : \tau_0 \mid \xi}{\Theta \mid \Gamma \vdash (x : \tau_1) \Rightarrow s : \tau_1 \to\xi\, \tau_0}\,[\textsc{Lam}] \quad \frac{\Theta \mid \Gamma, f : \tau_1 \to\xi\, \tau_0, x : \tau_1 \vdash s : \tau_0 \mid \xi}{\Theta \mid \Gamma \vdash \textsf{fix}\, f\,(x : \tau_1) \Rightarrow s : \tau_1 \to\xi\, \tau_0}\,[\textsc{Fix}]$$

$$\frac{\Theta, c : F[\xi] \mid \Gamma \vdash e : \tau}{\Theta \mid \Gamma \vdash [c : F[\xi]] \Rightarrow e : F[\xi] \to \tau}\,[\textsc{CapLam}] \quad \frac{\Theta \mid \Gamma \vdash e : F[\xi] \to \tau \qquad \Theta \mid \Gamma \vdash h : F[\xi]}{\Theta \mid \Gamma \vdash e[h] : \tau}\,[\textsc{CapApp}]$$

Statement Typing

$$\boxed{\Theta \mid \Gamma \vdash s : \tau \mid \xi} \qquad \frac{\Theta, c : F[\tau :: \xi] \mid \Gamma \mid \tau :: \xi \vdash s : \tau \qquad \Theta \mid \Gamma \vdash h : F[\tau :: \xi]}{\Theta \mid \Gamma \mid \xi \vdash \textsf{handle}\,\{\, c \Rightarrow s \,\}\,\textsf{with}\, h : \tau}\,[\textsc{Handle}]$$

$$\frac{\Theta \mid \Gamma \mid \xi \vdash s_0 : \tau_0 \qquad \Theta \mid \Gamma, x : \tau_0 \mid \xi \vdash s : \tau}{\Theta \mid \Gamma \mid \xi \vdash \textsf{val}\, x = s_0;\, s : \tau}\,[\textsc{Val}] \qquad \frac{\Theta \mid \Gamma \vdash e : \tau}{\Theta \mid \Gamma \mid \xi \vdash \textsf{return}\, e : \tau}\,[\textsc{Ret}]$$

$$\frac{\Theta \mid \Gamma \vdash e : \tau' \to\xi\, \tau \qquad \Theta \mid \Gamma \vdash e' : \tau'}{\Theta \mid \Gamma \mid \xi \vdash e(e') : \tau}\,[\textsc{App}] \quad \frac{\Theta \mid \Gamma \vdash h : F[\xi] \qquad \Sigma(F) = \tau' \to \tau \qquad \Theta \mid \Gamma \vdash e : \tau'}{\Theta \mid \Gamma \mid \xi \vdash \textsf{do}\, h(e) : \tau}\,[\textsc{Do}]$$

Capability Typing

$$\boxed{\Theta \mid \Gamma \vdash h : F[\xi]} \qquad \frac{\Theta \mid \Gamma \vdash h : F[\xi]}{\Theta \mid \Gamma \vdash \textsf{lift}\, h : F[\tau :: \xi]}\,[\textsc{CapLift}] \quad \frac{\Theta(c) = F[\xi]}{\Theta \mid \Gamma \vdash c : F[\xi]}\,[\textsc{CapVar}]$$

$$\frac{\Theta, k : \textsf{Resume}_i[\xi] \mid \Gamma, x : \tau_1 \vdash s : \tau \mid \xi \qquad \Sigma(F) = \tau_1 \to \tau_0 \qquad \Sigma(\textsf{Resume}_i) = \tau_0 \to \tau \qquad i \text{ fresh}}{\Theta \mid \Gamma \vdash \{\, (x, k) \Rightarrow s \,\} : F[\tau :: \xi]}\,[\textsc{CapHandler}]$$

Figure 5.3: Type system of λ_{Cap}.

CapLift adjusts a capability h typed against $F[\xi]$ to be compatible with a larger stack shape $F[\tau :: \xi]$. This is in spirit similar to *adaptors* in the language Frank [Convent et al., 2020], to *lift* in Helium [Biernacki et al., 2019a], and to *inject* in Koka [Leijen, 2018]. However, instead of adjusting arbitrary effectful expressions, we only perform the adjustments on capabilities. As we will see, this allows us to guarantee that the lifting itself is performed at compile time. Finally, rule CapHandler checks the body of a handler implementation $\{ (x, k) \Rightarrow s \}$ against a stack shape $\tau :: \xi$. A handler implementation for an effect operation F takes an argument of type τ_1 and a continuation k, which can be thought of as an effectful function $\tau_0 \rightarrow \xi \tau$. We model resumptions as effect operations. The body s of the handler is evaluated in stack shape ξ, that is, outside of the delimiter that introduced it.

5.2.3 Translation

In this subsection, we describe the translation of λ_{Cap} to simply-typed lambda calculus (STLC) [Barendregt, 1992], extended with a standard **letrec** operator to express **fix**. In the translation of λ_{Cap} capabilities are still present at runtime. Later, in the translation of $\boldsymbol{\lambda}_{\mathsf{Cap}}$ (Section 5.3.2), we will use a two-level lambda calculus as the target, marking some abstractions as static and others as dynamic. This allows us to prevent administrative redexes and, more importantly, to eliminate all abstractions related to handlers and capabilities.

Figure 5.4 defines the translation on types and mutually recursive translations of the different syntactic categories of terms. We translate programs to iterated CPS. Theorem 20 shows that our translation takes well-typed λ_{Cap} programs to well-typed STLC programs.

Target Language

The target of our translation is a call-by-value STLC extended with **letrec**, base types, and primitive operations. As usual, we write lambda abstraction as $\lambda x \Rightarrow e$, but use the infix notation $e_0 \mathbin{@} e_1$ for application [Nielson and Nielson, 1996]. We sometimes use **let** bindings in the target language assuming the standard shorthand: $\textbf{let}\, \mathsf{x} = \mathsf{e}\, \textbf{in}\, \mathsf{e}' \doteq (\lambda \mathsf{x} \Rightarrow \mathsf{e}') \mathbin{@} \mathsf{e}$.

Translation of Types

The translation of types $\mathcal{T}[\![\cdot]\!]$ maps base types to base types in STLC and effectful function types $\tau_1 \rightarrow \xi\, \tau_0$ to functions from τ_1 to effectful computations $\mathcal{C}[\![\tau_0]\!]_\xi$. Capability function types are translated to function types in the target language. Capability parameters of type $F[\xi]$ are translated like an effectful function of type $\tau_1 \rightarrow \xi\, \tau_0$ for $\Sigma(F) = \tau_1 \rightarrow \tau_0$.

The meta function $\mathcal{C}[\![\tau]\!]_\xi$ computes the type in STLC corresponding to an effectful computation with return type τ in stack shape ξ. Programs with an empty stack shape cannot use any control effects and consequently are not CPS translated. The translation of non-empty stack shapes recursively translates the rest of the stack shape.

$$
\begin{aligned}
&\mathcal{T}\llbracket\, \mathsf{Int} \,\rrbracket && = \ \mathsf{Int} \\
&\mathcal{T}\llbracket\, \tau_1 \to\xi\, \tau_0 \,\rrbracket && = \ \mathcal{T}\llbracket\, \tau_1 \,\rrbracket \to \mathcal{C}\llbracket\, \tau_0 \,\rrbracket_{\xi} \\
&\mathcal{T}\llbracket\, F[\xi] \to \tau \,\rrbracket && = \ \mathcal{T}\llbracket\, F[\xi] \,\rrbracket \to \mathcal{T}\llbracket\, \tau \,\rrbracket \\
&\mathcal{T}\llbracket\, F[\xi] \,\rrbracket && = \ \mathcal{T}\llbracket\, \tau_1 \,\rrbracket \to \mathcal{C}\llbracket\, \tau_0 \,\rrbracket_{\xi} \\
&\quad \text{where} \quad \Sigma(F) \ = \ \tau_1 \to \tau_0 && \\
&\mathcal{C}\llbracket\, \tau \,\rrbracket_{\emptyset} && = \ \mathcal{T}\llbracket\, \tau \,\rrbracket \\
&\mathcal{C}\llbracket\, \tau \,\rrbracket_{\tau_0 \,::\, \xi} && = \ (\mathcal{T}\llbracket\, \tau \,\rrbracket \to \mathcal{C}\llbracket\, \tau_0 \,\rrbracket_{\xi}) \to \mathcal{C}\llbracket\, \tau_0 \,\rrbracket_{\xi}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}\llbracket\, e_0(e_1) \,\rrbracket_{\xi} && = \ \mathcal{E}\llbracket\, e_0 \,\rrbracket \ @ \ \mathcal{E}\llbracket\, e_1 \,\rrbracket \\
&\mathcal{S}\llbracket\, \mathbf{val}\, x \ = \ s_0;\ s \,\rrbracket_{\bullet} && = \ \mathbf{let}\, x \ = \ \mathcal{S}\llbracket\, s_0 \,\rrbracket_{\bullet}\ \mathbf{in}\, \mathcal{S}\llbracket\, s \,\rrbracket_{\bullet} \\
&\mathcal{S}\llbracket\, \mathbf{val}\, x \ = \ s_0;\ s \,\rrbracket_{\tau \,::\, \xi} && = \ \lambda k \Rightarrow \mathcal{S}\llbracket\, s_0 \,\rrbracket_{\tau \,::\, \xi} \ @ \ (\lambda x \Rightarrow \mathcal{S}\llbracket\, s \,\rrbracket_{\tau \,::\, \xi} \ @ \ k) \\
&\mathcal{S}\llbracket\, \mathbf{return}\, e \,\rrbracket_{\bullet} && = \ \mathcal{E}\llbracket\, e \,\rrbracket \\
&\mathcal{S}\llbracket\, \mathbf{return}\, e \,\rrbracket_{\tau \,::\, \xi} && = \ \lambda k \Rightarrow k \ @ \ \ \mathcal{E}\llbracket\, e \,\rrbracket \\
&\mathcal{S}\llbracket\, \mathbf{do}\, h(e) \,\rrbracket_{\xi} && = \ \mathcal{H}\llbracket\, h \,\rrbracket_{\xi} \ @ \ \mathcal{E}\llbracket\, e \,\rrbracket \\
&\mathcal{S}\llbracket\, \mathbf{handle}\, \{\, c \Rightarrow s \,\}\, \mathbf{with}\, h \,\rrbracket_{\xi} && = \ (\lambda c \Rightarrow \mathcal{S}\llbracket\, s \,\rrbracket_{\tau \,::\, \xi}) \ @ \ (\mathcal{H}\llbracket\, h \,\rrbracket_{\tau \,::\, \xi}) \ @ \ (\lambda x \Rightarrow \mathcal{S}\llbracket\, \mathbf{return}\, x \,\rrbracket_{\xi}) \\
&\quad \text{where} \quad \Theta \mid \Gamma \mid \xi \vdash_{\mathsf{stm}} \ \mathbf{handle}\, \{\, c \Rightarrow s \,\}\, \mathbf{with}\, h \ : \ \tau &&
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\llbracket\, \mathsf{True} \,\rrbracket && = \ \mathsf{True} \\
&\mathcal{E}\llbracket\, x \,\rrbracket && = \ x \\
&\mathcal{E}\llbracket\, (x \ : \ \tau_1) \Rightarrow s \,\rrbracket && = \ \lambda x \Rightarrow \mathcal{S}\llbracket\, s \,\rrbracket_{\xi} \\
&\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{exp}} \ (x \ : \ \tau_1) \Rightarrow s \ : \ \tau_1 \to\xi\, \tau_0 && \\
&\mathcal{E}\llbracket\, \mathbf{fix}\, f\ (x \ : \ \tau_1) \Rightarrow s \,\rrbracket && = \ \mathbf{letrec}\, f \ = \ (\lambda x \Rightarrow \mathcal{S}\llbracket\, s \,\rrbracket_{\xi})\, \mathbf{in}\, f \\
&\quad \text{where}\ \Theta \mid \Gamma \vdash_{\mathsf{exp}} \ \mathbf{fix}\, f\ (x \ : \ \tau_1) \Rightarrow s \ : \ \tau_1 \to\xi\, \tau_0 && \\
&\mathcal{E}\llbracket\, [c \ : \ F[\xi]] \Rightarrow e \,\rrbracket && = \ \lambda c \Rightarrow \mathcal{E}\llbracket\, e \,\rrbracket \\
&\mathcal{E}\llbracket\, e[h] \,\rrbracket && = \ \mathcal{E}\llbracket\, e \,\rrbracket \ @ \ \mathcal{H}\llbracket\, h \,\rrbracket_{\xi} \\
&\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{cap}} \ h \ : \ F[\xi] &&
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{H}\llbracket\, c \,\rrbracket_{\xi} && = \ c \\
&\mathcal{H}\llbracket\, \{\, (x,\ k) \Rightarrow s \,\} \,\rrbracket_{\tau \,::\, \xi} && = \ \lambda x \Rightarrow \lambda k \Rightarrow \mathcal{S}\llbracket\, s \,\rrbracket_{\xi} \\
&\mathcal{H}\llbracket\, \mathbf{lift}\, h \,\rrbracket_{\tau \,::\, \bullet} && = \ \lambda x \Rightarrow \lambda k \Rightarrow k \ @ \ (\mathcal{H}\llbracket\, h \,\rrbracket_{\bullet} \ @ \ x) \\
&\mathcal{H}\llbracket\, \mathbf{lift}\, h \,\rrbracket_{\tau_0 \,::\, \tau_1 \,::\, \xi} && = \ \lambda x \Rightarrow \lambda k \Rightarrow \lambda j \Rightarrow \mathcal{H}\llbracket\, h \,\rrbracket_{\tau_1 \,::\, \xi} \ @ \ x \ @ \ (\lambda y \Rightarrow k \ @ \ y \ @ \ j)
\end{aligned}
$$

Figure 5.4: Translation of λ_{Cap} to STLC.

It adds one layer of CPS translation with this recursively translated type as the answer type. For example, we have the following translations:

$$\begin{aligned}
\mathcal{C}[\![\, \mathsf{Bool} \,]\!]_{\bullet} &\doteq \mathsf{Bool} \\
\mathcal{C}[\![\, \mathsf{Bool} \,]\!]_{[\mathsf{Int}]} &\doteq (\mathsf{Bool} \to \mathsf{Int}) \to \mathsf{Int} \\
\mathcal{C}[\![\, \mathsf{Bool} \,]\!]_{[\mathsf{Int},\, \mathsf{String}]} &\doteq (\mathsf{Bool} \to \mathcal{C}[\![\, \mathsf{Int} \,]\!]_{[\mathsf{String}]}) \to \mathcal{C}[\![\, \mathsf{Int} \,]\!]_{[\mathsf{String}]} \\
&\doteq (\mathsf{Bool} \to ((\mathsf{Int} \to \mathsf{String}) \to \mathsf{String})) \to ((\mathsf{Int} \to \mathsf{String}) \to \mathsf{String})
\end{aligned}$$

We can see that our translation performs a CPS transformation for each entry in the stack shape ξ.

Translation of Statements

The translation of statements $\mathcal{S}[\![\, s \,]\!]_{\xi}$ is indexed by a stack shape ξ. Source statements s with return type τ in stack shape ξ are translated to effectful computations of type $\mathcal{C}[\![\, \tau \,]\!]_{\xi}$. In the case of a pure statement without effects, the stack shape is empty and we do not perform a CPS translation. Returning translates to just the returned expression and, to preserve sharing of results, sequencing translates to a let binding. In the case where the stack shape is non-empty, we perform a single layer of CPS-translation. We translate the use of capabilities ($\mathbf{do}\, h(e)$) to function applications. The translation of $\mathbf{handle}\, \{\, c \Rightarrow s \,\}\, \mathbf{with}\, h$ binds c to the translation of h in the translated body s. Importantly, it also delimits effects by applying the translated body to the empty continuation.

Translation of Expressions

In the translation of expressions, we map capability abstraction to ordinary function abstraction and capability application to ordinary function application. The translation of function abstraction and capability application is type directed: the stack shape ξ guides the translation of the function body and the handler, respectively.

Translation of Capabilities

To translate handler implementations, the body s is translated as a statement with a smaller stack shape ξ. This models the fact that the handler implementation is evaluated outside of the delimiter it introduces. The translation of a handler implementation is only defined for non-empty stack shapes $\tau :: \xi$. Our typing rules make sure that this is always the case. The translation of lifted capabilities looks a bit involved. The goal is to make capability h, typed against a stack shape ξ, usable with an extended stack shape $\tau :: \xi$. Since the number of elements in the stack shape corresponds to the number of continuation arguments, we have to adapt the capability to take one more continuation. In the case of a stack shape with at least two elements, the translation abstracts over the argument x and the first two continuations k and j. It then applies the translated capability to the argument and a single continuation that is the composition of k and j. The case of a singleton stack shape never occurs in a closed well-typed program and is purely listed for our formalization.

Example Let us translate the following example in the empty stack shape:

$$\mathcal{S}[\![\, \textsf{handle}\ \{\ c \Rightarrow \textsf{val}\ x\ =\ \textsf{do}\ c(\mathsf{True});\ \textsf{return}\ e\ \}\ \textsf{with}\ h \,]\!]_\bullet$$

Assuming an answer type of Int, we obtain:

$$\underbrace{(\lambda c \Rightarrow \mathcal{S}[\![\ \textsf{val}\ x\ =\ \textsf{do}\ c(\mathsf{True});\ \textsf{return}\ e\ ]\!]_{\mathsf{Int}}}_{\lambda k \Rightarrow \mathcal{S}[\![\ \underbrace{\textsf{do}\ c(\mathsf{True})}_{c\ @\ \mathsf{True}}\ ]\!]_{\mathsf{Int}}\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\ \underbrace{\textsf{return}\ e}_{\lambda j \Rightarrow j\ @\ \mathcal{E}[\![\, e \,]\!]}\ ]\!]_{\mathsf{Int}}\ @\ k)})\ @\ (\mathcal{H}[\![\, h \,]\!]_{\mathsf{Int}})\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\, \textsf{return}\ x \,]\!]_\bullet)$$

By $\mathcal{S}[\![\, \textsf{return}\ x \,]\!]_\bullet\ =\ x$, the overall example translates to:

$$(\lambda c \Rightarrow \lambda k \Rightarrow c\ @\ \mathsf{True}\ @\ (\lambda x \Rightarrow (\lambda j \Rightarrow j\ @\ \mathcal{E}[\![\, e \,]\!])\ @\ k))\ @\ (\mathcal{H}[\![\, h \,]\!]_{\mathsf{Int}})\ @\ (\lambda x \Rightarrow x)$$

This illustrates that capability passing translates to normal function abstraction and application and that we support control effects by translating to iterated CPS.

5.3 The Language $\mathbb{\lambda}_{\mathsf{Cap}}$

We now refine λ_{Cap} to a sub-language $\mathbb{\lambda}_{\mathsf{Cap}}$. On this sub-language we are able to fully eliminate the overhead introduced by handler abstractions. We present a second translation, this time to 2-level lambda calculus. This allows us to prove that all abstractions and applications related to effect handlers will be statically reduced at compile time.

5.3.1 Syntax and Typing

Figure 5.5 lists the syntax of types of $\mathbb{\lambda}_{\mathsf{Cap}}$. The syntax of terms is exactly the same as the one for λ_{Cap}. In the syntax of types we now distinguish between *dynamic types* and *static types*, similar to the syntactic separation of expressions and capabilities (this difference is highlighted in gray). Static types are sequences of capability parameters ending in a dynamic type, which ensures that all capability arguments come before any other arguments. This is the only difference to λ_{Cap} in Figure 5.2. Later in this section, we will see that terms of static types will be eliminated (due to inlining and specialization) during translation while terms of dynamic types will appear in the generated program.

Changes to the typing rules for $\mathbb{\lambda}_{\mathsf{Cap}}$ (Figure 5.5) compared to λ_{Cap} (Figure 5.3) are also highlighted in gray. Importantly, while the judgement form $\Theta \mid \Gamma \vdash\ e : \sigma$ may assign a static type σ to expressions, the typing rules for statements and capabilities remain unchanged. Their premises still require expressions to be typed against a dynamic type τ. This way, we make sure that all effectful functions are always fully applied to the corresponding capabilities.

We treat capabilities as second class [Osvald et al., 2016]. That is, they cannot be returned from a function. This becomes evident in the typing rules. In the rule Ret (Figure 5.3):

Dynamic Types

$$\begin{array}{lll} \tau & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \ldots & \text{base types} \\ & \mid & \tau \to \xi\, \tau & \text{effectful function type} \end{array}$$

Static Types

$$\begin{array}{lll} \sigma & ::= & F[\xi] \to \sigma & \text{capability function type} \\ & \mid & \tau & \text{dynamic type} \end{array}$$

Operation Names

$$F \;::=\; \mathsf{Fork} \mid \mathsf{Fail} \mid \mathsf{Emit} \mid \mathsf{Resume}_i \mid \ldots$$

Operation Signatures

$$\Sigma \;::=\; \emptyset \mid \Sigma,\, F : \tau_1 \to \tau_0$$

Type Environment

$$\Gamma \;::=\; \emptyset \mid \Gamma,\, x : \tau$$

Capability Environment

$$\Theta \;::=\; \emptyset \mid \Theta,\, c : F[\xi]$$

Stack Shape

$$\xi \;::=\; \bullet \mid \tau :: \xi$$

Expression Typing

$$\boxed{\Theta \mid \Gamma \vdash e : \sigma}$$

$$\frac{\Gamma(x) = \tau}{\Theta \mid \Gamma \vdash x : \tau}\;[\textsc{Var}]$$

$$\frac{\Theta \mid \Gamma,\, x : \tau_1 \mid \xi \vdash s : \tau_0}{\Theta \mid \Gamma \vdash (x : \tau_1) \Rightarrow s : \tau_1 \to \xi\, \tau_0}\;[\textsc{Lam}] \qquad \frac{\Theta \mid \Gamma,\, f : \tau_1 \to \xi\, \tau_0,\, x : \tau_1 \mid \xi \vdash s : \tau_0}{\Theta \mid \Gamma \vdash \mathbf{fix}\, f\,(x : \tau_1) \Rightarrow s : \tau_1 \to \xi\, \tau_0}\;[\textsc{Fix}]$$

$$\frac{\Theta,\, c : F[\xi] \mid \Gamma \vdash e : \sigma}{\Theta \mid \Gamma \vdash [c : F[\xi]\,] \Rightarrow e : F[\xi] \to \sigma}\;[\textsc{CapLam}] \qquad \frac{\Theta \mid \Gamma \vdash e : F[\xi] \to \sigma \qquad \Theta \mid \Gamma \vdash h : F[\xi]}{\Theta \mid \Gamma \vdash e[h] : \sigma}\;[\textsc{CapApp}]$$

Figure 5.5: Difference in syntax of types and expression typing rules for $\boldsymbol{\lambda}_{\mathsf{Cap}}$.

$$\frac{\Theta \mid \Gamma \vdash e : \tau}{\Theta \mid \Gamma \mid \xi \vdash \mathbf{return}\, e : \tau}\;[\textsc{Ret}]$$

The returned pure expression e has to be typed against a dynamic type τ. Expressions thus are always fully specialized (that is, applied to capabilities) before they can be returned. Similarly, argument expressions in rule App are required to be of a dynamic type τ_1, which means that we cannot abstract over capability abstractions.

We model resumptions as capabilities, which makes them second class. This is in order to guarantee full elimination of the handler abstraction at compile time. Since capabilities will be inlined, so will resumptions.

5.3.2 Translation

The programs generated by the translation of λ_{Cap} in Figure 5.4 abstract over handler implementations and pass them along at runtime. In the translation of $\boldsymbol{\lambda}_{\mathsf{Cap}}$, we avoid this passing of capabilities at run time and statically specialize functions to the capabilities that they use. Maybe more importantly, we also specialize the inlined capabilities to the *context* they are used in. This enables optimizations across effect calls.

We want to guarantee that certain redexes never occur in the generated program. In particular, there are two classes of redexes that we want to avoid. Firstly, we avoid generating administrative beta redexes in our CPS translation. This standard use of multi-level lambda calculi in the translation of control operators has been introduced by Danvy and Filinski [1992]. Even though not listed in Figure 5.4, in our benchmarks we also do this for the unrestricted language in Figure 5.2. Secondly, we avoid generating redexes associated with the effect handler abstraction. The significant contribution is that handling effects, calling effect operations, and lifting capabilities does not introduce *any* redexes in the generated program.

Figure 5.6 presents the refined translation from λ_{Cap} to 2-level lambda calculus [Taha and Sheard, 2000, Nielson and Nielson, 1996, Jones et al., 1993]. The translation is the same as the one in Figure 5.4 except for annotations to distinguish static from dynamic program fragments. The annotations are automatically inserted as part of the definition of our translation. In Theorem 22 we prove "stage-time correctness", *i.e.*, that we never confuse static and dynamic functions. This is only possible because the type system of λ_{Cap} restricted the use of capabilities, making them second class.

2-level Lambda Calculus

The general idea of multi-level lambda calculi [Nielson and Nielson, 1996] is to mark some abstractions and applications as static and some as residual. Static redexes will be reduced during translation, while residual redexes will be generated, that is, residualized. We adopt the terminology of Taha and Sheard [1997] and refer to the annotations as *staging annotations*. We use standard notation that we briefly review. On the type level we use red color and an underline for types of *residual* terms (*i.e.* terms that will be residualized). For example $\underline{\mathsf{Int}} \underline{\to} \underline{\mathsf{Int}}$ is the type of a generated function from integers to integers. We write types of *static* (*i.e.* stage time) terms in blue with an overbar. For example $\underline{\mathsf{Int}} \overline{\Rightarrow} \underline{\mathsf{Int}}$ is the type of a static function between residualized integers. Similarly, on the term level we write residual terms in red with an underline. For example, $\underline{1} \underline{+} \underline{2}$ is the term that adds the integer one and the integer two. This redex will occur in the generated program. We write terms that we evaluate during translation in blue with an overbar. For example $(\overline{\lambda x \Rightarrow} x)\ \overline{@}\ \underline{5}$ will statically evaluate to the term $\underline{5}$. We use $\underline{\mathcal{C}}[\![\, \tau \,]\!]_{\xi}$ to describe the type of residual effectful computations. The whole computation type $\mathcal{C}[\![\, \tau \,]\!]_{\xi}$ as defined in Figure 5.4 is residualized. The type $\overline{\mathcal{C}}[\![\, \tau \,]\!]_{\xi}$ represents static computations. Importantly, while the answer types are residual (*e.g.*, $\underline{\tau}$) the structure of the computation is static.

Reify and Reflect

To mediate between residual effectful computations and static effectful computations, we define two mutually recursive meta functions Reify and Reflect.

$$
\begin{array}{lcl}
\mathcal{T}\llbracket \mathsf{Int} \rrbracket & = & \underline{\mathsf{Int}} \\
\mathcal{T}\llbracket \tau_1 \to \xi\, \tau_0 \rrbracket & = & \mathcal{T}\llbracket \tau_1 \rrbracket \,\underline{\to}\, \underline{\mathcal{C}}\llbracket \tau_0 \rrbracket_{\xi} \\[1ex]
\mathcal{T}\llbracket F[\xi] \to \sigma \rrbracket & = & \mathcal{T}\llbracket F[\xi] \rrbracket \,\overline{\Rightarrow}\, \mathcal{T}\llbracket \sigma \rrbracket \\
\mathcal{T}\llbracket F[\xi] \rrbracket & = & \mathcal{T}\llbracket \tau_1 \rrbracket \,\overline{\Rightarrow}\, \overline{\mathcal{C}}\llbracket \tau_0 \rrbracket_{\xi} \\
\quad \text{where} \quad \Sigma(F) \;=\; \tau_1 \to \tau_0 & & \\[1ex]
\underline{\mathcal{C}}\llbracket \tau \rrbracket_{\bullet} & = & \mathcal{T}\llbracket \tau \rrbracket \\
\underline{\mathcal{C}}\llbracket \tau \rrbracket_{\tau_0 \,::\, \xi} & = & (\mathcal{T}\llbracket \tau \rrbracket \,\underline{\to}\, \underline{\mathcal{C}}\llbracket \tau_0 \rrbracket_{\xi}) \,\underline{\to}\, \underline{\mathcal{C}}\llbracket \tau_0 \rrbracket_{\xi} \\[1ex]
\overline{\mathcal{C}}\llbracket \tau \rrbracket_{\bullet} & = & \mathcal{T}\llbracket \tau \rrbracket \\
\overline{\mathcal{C}}\llbracket \tau \rrbracket_{\tau_0 \,::\, \xi} & = & (\mathcal{T}\llbracket \tau \rrbracket \,\overline{\Rightarrow}\, \overline{\mathcal{C}}\llbracket \tau_0 \rrbracket_{\xi}) \,\overline{\Rightarrow}\, \overline{\mathcal{C}}\llbracket \tau_0 \rrbracket_{\xi} \\[2ex]
\mathcal{S}\llbracket e_0(e_1) \rrbracket_{\xi} & = & \textsc{Reflect}_{\xi}\;\; (\mathcal{E}\llbracket e_0 \rrbracket \,\underline{@}\, \mathcal{E}\llbracket e_1 \rrbracket) \\[1ex]
\mathcal{S}\llbracket \mathbf{val}\, x \;=\; s_0;\, s \rrbracket_{\bullet} & = & \underline{\mathsf{let}}\; x \;=\; \mathcal{S}\llbracket s_0 \rrbracket_{\bullet}\;\; \underline{\mathsf{in}}\; \mathcal{S}\llbracket s \rrbracket_{\bullet} \\
\mathcal{S}\llbracket \mathbf{val}\, x \;=\; s_0;\, s \rrbracket_{\tau \,::\, \xi} & = & \overline{\lambda k \Rightarrow} \mathcal{S}\llbracket s_0 \rrbracket_{\tau \,::\, \xi} \;\overline{@}\; (\overline{\lambda x \Rightarrow} \mathcal{S}\llbracket s \rrbracket_{\tau \,::\, \xi} \;\overline{@}\; k) \\[1ex]
\mathcal{S}\llbracket \mathbf{return}\, e \rrbracket_{\bullet} & = & \mathcal{E}\llbracket e \rrbracket \\
\mathcal{S}\llbracket \mathbf{return}\, e \rrbracket_{\tau \,::\, \xi} & = & \overline{\lambda k \Rightarrow} k \;\overline{@}\;\; \mathcal{E}\llbracket e \rrbracket \\[1ex]
\mathcal{S}\llbracket \mathbf{do}\, h(e) \rrbracket_{\xi} & = & \mathcal{H}\llbracket h \rrbracket \;\overline{@}\; \mathcal{E}\llbracket e \rrbracket \\[1ex]
\mathcal{S}\llbracket \mathbf{handle}\,\{\, c \Rightarrow s \,\}\, \mathbf{with}\, h \rrbracket_{\xi} & = & (\overline{\lambda c \Rightarrow} \mathcal{S}\llbracket s \rrbracket_{\tau \,::\, \xi}) \;\overline{@}\; (\mathcal{H}\llbracket h \rrbracket_{\tau \,::\, \xi}) \;\overline{@}\; (\overline{\lambda x \Rightarrow} \mathcal{S}\llbracket \mathbf{return}\, x \rrbracket_{\xi}) \\
\quad \text{where} \quad \Theta \mid \Gamma \mid \xi \vdash_{\mathsf{stm}} \mathbf{handle}\,\{\, c \Rightarrow s \,\}\, \mathbf{with}\, h \;:\; \tau & & \\[2ex]
\mathcal{E}\llbracket \mathsf{True} \rrbracket & = & \underline{\mathsf{True}} \\
\mathcal{E}\llbracket x \rrbracket & = & x \\
\mathcal{E}\llbracket (x \;:\; \tau_1) \Rightarrow s \rrbracket & = & \underline{\lambda x \Rightarrow} \textsc{Reify}_{\xi}\;\; \mathcal{S}\llbracket s \rrbracket_{\xi} \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{exp}} (x \;:\; \tau_1) \Rightarrow s \;:\; \tau_1 \to \xi\, \tau_0 & & \\[1ex]
\mathcal{E}\llbracket \mathbf{fix}\, f\; (x \;:\; \tau_1) \Rightarrow s \rrbracket & = & \underline{\mathsf{letrec}}\, f \;=\; (\underline{\lambda x \Rightarrow} \textsc{Reify}_{\xi}\, \mathcal{S}\llbracket s \rrbracket_{\xi})\;\; \underline{\mathsf{in}}\; f \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{exp}} \mathbf{fix}\, f\; (x \;:\; \tau_1) \Rightarrow s \;:\; \tau_1 \to \xi\, \tau_0 & & \\[1ex]
\mathcal{E}\llbracket [c \;:\; F[\xi]] \Rightarrow e \rrbracket & = & \overline{\lambda c \Rightarrow} \mathcal{E}\llbracket e \rrbracket \\
\mathcal{E}\llbracket e[h] \rrbracket & = & \mathcal{E}\llbracket e \rrbracket \;\overline{@}\; \mathcal{H}\llbracket h \rrbracket_{\xi} \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{cap}} h \;:\; F[\xi] & & \\[2ex]
\mathcal{H}\llbracket c \rrbracket_{\xi} & = & c \\[1ex]
\mathcal{H}\llbracket \{\, (x,\, k) \Rightarrow s \,\} \rrbracket_{\tau \,::\, \xi} & = & \overline{\lambda x \Rightarrow}\, \overline{\lambda k \Rightarrow} \mathcal{S}\llbracket s \rrbracket_{\xi} \\[1ex]
\mathcal{H}\llbracket \mathbf{lift}\, h \rrbracket_{\tau \,::\, \bullet} & = & \overline{\lambda x \Rightarrow}\, \overline{\lambda k \Rightarrow} k \;\overline{@}\; (\mathcal{H}\llbracket h \rrbracket_{\bullet} \;\overline{@}\; x) \\
\mathcal{H}\llbracket \mathbf{lift}\, h \rrbracket_{\tau_0 \,::\, \tau_1 \,::\, \xi} & = & \overline{\lambda x \Rightarrow}\, \overline{\lambda k \Rightarrow}\, \overline{\lambda j \Rightarrow} \mathcal{H}\llbracket h \rrbracket_{\tau_1 \,::\, \xi} \;\overline{@}\; x \;\overline{@}\; (\overline{\lambda y \Rightarrow} k \;\overline{@}\; y \;\overline{@}\; j)
\end{array}
$$

Figure 5.6: Translation of λ_{Cap} to 2-level lambda calculus.

$$
\begin{array}{lcl}
\textsc{Reify}_{\xi} & : & \overline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi} \to \underline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi} \\
\textsc{Reify}_{\bullet}\; s & \doteq & s \\
\textsc{Reify}_{(\tau\;::\;\xi)}\; s & \doteq & \underline{\lambda k \Rightarrow} \textsc{Reify}_{\xi}\; (s \;\overline{@}\; (\overline{\lambda x \Rightarrow} \textsc{Reflect}_{\xi}\; (k \;\underline{@}\; x))) \\
\\
\textsc{Reflect}_{\xi} & : & \underline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi} \to \overline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi} \\
\textsc{Reflect}_{\bullet}\; s & \doteq & s \\
\textsc{Reflect}_{(\tau\;::\;\xi)}\; s & \doteq & \overline{\lambda k \Rightarrow} \textsc{Reflect}_{\xi}\; (s \;\underline{@}\; (\underline{\lambda x \Rightarrow} \textsc{Reify}_{\xi}\; (k \;\overline{@}\; x)))
\end{array}
$$

The meta function REIFY converts a static computation of type $\overline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi}$ to a residual computation of type $\underline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi}$. In other words, it residualizes the statement. It is defined by induction over the stack shape, introducing one continuation argument for every type in the stack shape. Dually, the meta function REFLECT converts a residual computation of type $\underline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi}$ to a static computation of type $\overline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi}$. For every type in the stack shape, it generates one application to a reified continuation. This way, functions always abstract over and are always applied to all arguments and continuations.

Translation of Statements

We translate statements typed in a stack shape ξ with result of type τ to *static* computations of type $\overline{\mathcal{C}}[\![\,\tau\,]\!]_{\xi}$. We want to preserve function applications, so we generate an application and reflect the resulting statement. To preserve sharing, we translate sequenced pure statements to a residual let binding. When translating sequencing and returning of effectful statements, we mark all continuation abstractions and applications as static. This allows us to avoid administrative beta-redexes. We translate the binding of capability variables in handlers and the use of capabilities to static binding and application. This ensures that capabilities are fully inlined at their call-site.

Translation of Expressions

We always translate constants, variables and effectful functions to residual terms. The translation of effectful functions and effectful recursive functions requires us to reify function bodies. While we do not reduce function applications present in the original program, we want to perform capability passing at compile time. Therefore, we translate capability functions to static abstractions and capability application to static application. This ensures that they are reduced at compile time and no redexes involving capability passing will be generated.

Translation of Capabilities

Handler implementations translate to static functions that take a static argument and a static continuation. In contrast to effectful functions, we do not reify the bodies of handler implementations. This way, the context of a call to a capability will be inlined into the handler implementation, which leads to the optimization across effect operations that we want to achieve. Lifting a capability to be compatible with a larger stack shape is fully static as well: the composition of contexts is performed at compile time. By inspecting the translation of **do**, **handle** and handlers, we can observe that they only introduce static abstractions and applications. The translation is designed to not generate any redexes associated with effect handlers.

Example Applying the translation to 2-level lambda calculus to the example from Section 5.2.3, we obtain

$$(\overline{\lambda c \Rightarrow}\, \overline{\lambda k \Rightarrow}\, c \mathbin{\overline{@}} \underline{\mathsf{True}} \mathbin{\overline{@}} (\overline{\lambda x \Rightarrow}\, (\overline{\lambda k' \Rightarrow}\, k' \mathbin{\overline{@}} \mathcal{E}[\![\, e \,]\!]) \mathbin{\overline{@}} k)) \mathbin{\overline{@}} (\mathcal{H}[\![\, h \,]\!]_{[\mathsf{Int}]}) \mathbin{\overline{@}} (\overline{\lambda x \Rightarrow}\, x)$$

which reduces statically to:

$$\mathcal{H}[\![\, h \,]\!]_{[\mathsf{Int}]} \mathbin{\overline{@}} \underline{\mathsf{True}} \mathbin{\overline{@}} (\overline{\lambda x \Rightarrow}\, \mathcal{E}[\![\, e \,]\!])$$

This illustrates that the handler implementation is inlined at the position of the call to the effect operation. Furthermore, the continuation $\overline{\lambda x \Rightarrow}\, \mathcal{E}[\![\, e \,]\!]$ will be inlined into the handler implementation at compile time.

Example We specialize recursive functions to the handler implementations that they use at their call-site. For example, we translate the expression

$$\mathcal{E}[\![\, [\mathsf{h} : \mathsf{Fail}[\,\mathsf{Int}\,]\,] \Rightarrow \textbf{fix}\, \mathsf{f}\,(\mathsf{x} : \mathsf{Int}) \Rightarrow \textbf{val}\, \mathsf{y} = \textbf{do}\, \mathsf{h}();\ \mathsf{f}(\mathsf{x}) \,]\!]$$

to the following static capability abstraction:

$$\begin{array}{l}
\overline{\lambda \mathsf{h} \Rightarrow}\, \underline{\mathsf{letrec}}\, \mathsf{f} = \underline{\lambda \mathsf{x} \Rightarrow} \\
\quad\quad \textsc{Reify}_{\mathsf{Int}}\, (\overline{\lambda \mathsf{k} \Rightarrow}\, \mathsf{h} \mathbin{\overline{@}} () \mathbin{\overline{@}} (\overline{\lambda \mathsf{y} \Rightarrow}\, (\textsc{Reflect}_{\mathsf{Int}}\, (\mathsf{f} \mathbin{\underline{@}} \mathsf{x})) \mathbin{\overline{@}} \mathsf{k})) \\
\quad \underline{\mathsf{in}}\, \mathsf{f}
\end{array}$$

At the call-site, the translated function will be statically applied to a capability. This way, the function and all its recursive calls will be specialized to this capability. This also entails that the recursive call can only occur in a context with the *same* stack shape and the *same* capabilities.

5.3.3 Abstracting over Handlers

Other than in λ_{Cap}, capabilities in $\boldsymbol{\lambda}_{\mathsf{Cap}}$ are *second class* [Osvald et al., 2016]. This allows us to prove that they never appear in translated programs, but prevents us from writing certain kinds of programs in $\boldsymbol{\lambda}_{\mathsf{Cap}}$. In particular, we cannot abstract over handlers as handler functions, a common idiom in Koka for example. Consider the following example written in λ_{Cap} following this idiom:

```
def handleFailList(prog : Fail[ IntList ] → Unit →[IntList] Int) {
  handle { fail ⇒ prog[fail]() } with { (u, k) ⇒ Nil }
}
```

We define a handler function that handles the Fail effect by discarding the continuation and answering with the empty list. This handler is useful and such a definition might be part of the standard library. However, this example is ruled out by the more restrictive type system of $\boldsymbol{\lambda}_{\mathsf{Cap}}$. Being a parameter, prog has a dynamic type, but capability application $\mathsf{prog}[\mathsf{fail}]$ demands that prog has a static type. We thus cannot define handlers as handler functions in $\boldsymbol{\lambda}_{\mathsf{Cap}}$.

We can, however, still define and reuse handlers in multiple places. Consider the following example using a hypothetical language construct **defhandler**:

$$
\begin{array}{l}
\textbf{defhandler}\ \mathsf{handleFailList}\ =\ \{\, ((),\ \mathsf{k}) \Rightarrow \mathsf{Nil}\,\}\ \textbf{in} \\
\ldots \\
\textbf{handle}\ \{\, \mathsf{fail} \Rightarrow \ldots \,\}\ \textbf{with}\ \mathsf{handleFailList} \\
\ldots
\end{array}
$$

This language construct is macro-expressible in $\boldsymbol{\lambda}_{\mathsf{Cap}}$ as

$$\textbf{defhandler}\ \mathsf{c}\ =\ \mathsf{h}\ \textbf{in}\ \mathsf{e}\ \doteq\ ([\mathsf{c}] \Rightarrow \mathsf{e})\ \mathsf{h}$$

Because capability abstractions are reduced statically, the newly defined handler will be inlined at all of its call sites, maintaining the guarantee that functions are specialized to the effect handlers.

5.4 Evaluation

We implemented λ_{Cap} and $\boldsymbol{\lambda}_{\mathsf{Cap}}$ as *shallow embeddings* into the dependently typed programming language Idris [Brady, 2013]. We use typed HOAS [Pfenning and Elliot, 1988] and represent the AST of residualized programs of type $\underline{\tau}$ as values of a data type indexed by the type τ. We use the host language Idris to both express source programs, as well as to express static abstractions and applications. For example, the type $\underline{\mathsf{Int}}\ \underline{\to}\ \underline{\mathsf{Int}}$ would correspond to the Idris type `Exp (Int → Int)` and the type $\underline{\mathsf{Int}} \Rightarrow \underline{\mathsf{Int}}$ would correspond to the Idris type `Exp Int → Exp Int`. Throughout our implementation, we use dependent types to index source and target programs by their types, including stack shapes, which we represent as a type-level list of types. For example the λ_{Cap} type $\mathsf{Int} \to [\mathsf{String}, \mathsf{Int}]\ \mathsf{Bool}$ would correspond to the Idris type `Exp (Int → Stm [String,Int] Bool)`. Our translation follows the inductive structure of this type-level list.

5.4.1 Theoretical Results

Our translations satisfy a few meta-theoretic properties. In our implementation we were careful to make these properties hold by construction. Firstly, our unstaged translation of λ_{Cap} (as presented in Figure 5.4) preserves well-typedness.

Theorem 20 (Typability of translated terms – unstaged)**.**

$$
\begin{array}{lll}
\Theta \mid \Gamma \mid \xi \vdash\ s : \tau & \text{implies}\ \mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{S}[\![\, s \,]\!]_{\xi} & : \mathcal{C}[\![\, \tau \,]\!]_{\xi} \\
\Theta \mid \Gamma \vdash\ h : F[\xi] & \text{implies}\ \mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{H}[\![\, h \,]\!] & : \mathcal{T}[\![\, F[\xi] \,]\!] \\
\Theta \mid \Gamma \vdash\ e : \tau & \text{implies}\ \mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{E}[\![\, e \,]\!] & : \mathcal{T}[\![\, \tau \,]\!]
\end{array}
$$

Proof. By induction over the typing derivations and case distinction on the stack shapes. □

Importantly, we obtain effect safety as corollary.

Corollary 21 (Effect safety)**.** *Given a closed statement s, if $\emptyset \mid \emptyset \mid \bullet \vdash s : \tau$, then evaluating $\mathcal{S}[\![s]\!]_{\bullet}$ will not get stuck.*

Effect safety immediately follows from Theorem 20 and soundness of STLC. Well-typedness is also preserved by the staged translation (Figure 5.6).

Theorem 22 (Typability of translated terms – staged)**.**

$$\begin{array}{llll}
\Theta \mid \Gamma \mid \xi \vdash s : \tau & \text{implies } \mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{S}[\![s]\!]_{\xi} & : \overline{\mathcal{C}}[\![\tau]\!]_{\xi} \\
\Theta \mid \Gamma \vdash h : F[\xi] & \text{implies } \mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{H}[\![h]\!] & : \mathcal{T}[\![F[\xi]]\!] \\
\Theta \mid \Gamma \vdash e : \sigma & \text{implies } \mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{E}[\![e]\!] & : \mathcal{T}[\![\sigma]\!]
\end{array}$$

Proof. By induction over the typing derivations and case distinction on the stack shapes. □

Our translation thus takes well-typed source programs to well-typed 2-level lambda calculus programs. From Theorem 22 and soundness of the 2-level lambda calculus follows *stage-time correctness*: the translation only applies static functions statically and residualizes applications of residual functions. In our implementation, we ensure this by distinguishing static and residual expressions on the type level.

Stage time correctness means that code generation does not fail for well-typed programs:

Theorem 23 (Full residualization)**.** *Given a closed statement s, if $\emptyset \mid \emptyset \mid \bullet \vdash s : \tau$ then its translation $\mathcal{S}[\![s]\!]_{\bullet}$ can be fully reduced to a residualized term.*

Proof. By Theorem 22, we have that $\vdash \mathcal{S}[\![s]\!]_{\bullet} : \overline{\mathcal{C}}[\![\tau]\!]_{\bullet}$. Now $\overline{\mathcal{C}}[\![\tau]\!]_{\bullet} = \mathcal{T}[\![\tau]\!]$. By induction on the rules of $\mathcal{T}[\![\cdot]\!]$, we get that translation of dynamic types τ results in a residual type $\underline{\tau'}$. Soundness of the 2-level lambda calculus guarantees that stage-time reduction will not get stuck. Our translation does not introduce any static **letrec**, which guarantees termination. □

By Corollary 23 and soundness of the 2-level lambda calculus, it is easy to see that effect safety (Corollary 21) also extends to the staged translation. That is, reducing the residualized term will not get stuck.

Our careful separation of capability abstractions and applications from function abstractions and applications in λ_{Cap} allows us to guarantee that abstracting over effect operations with handlers does not incur any runtime overhead.

Theorem 24 (Full elimination)**.** *The translations of capability abstraction, capability application,* **do** $h(e)$, **handle** $\{ c \Rightarrow s \}$ **with** h, $\{ (x, k) \Rightarrow s \}$, *and* **lift** h *do not introduce any residual lambda abstractions or applications, except for those in the translation of their subterms.*

Proof. By inspection of our translation with staging annotations in Figure 5.6. All abstractions and applications that the translations immediately introduce are marked as static. □

In particular, capability passing is performed statically, handlers are fully inlined, local continuations are fully inlined, and continuations at the call-site of effect operations are inlined in the (already inlined) handler implementations.

While our translation guarantees the elimination of effect handlers, there is still a cost that originates from the use of control effects. Handled statements are translated with one more element in the stack shape. To support continuation capture, effectful function abstractions are CPS transformed and receive one additional continuation argument per stack shape entry, that is, for every enclosing handler. In other words, the only additional cost per handler is induced by the number of continuation arguments and materializes in Reify and Reflect.

5.4.2 Performance Results

We assess the performance of the code generated from λ_{Cap} and $\mathbb{\lambda}_{\mathsf{Cap}}$ and compare it to existing languages with effect handlers and control effects. The benchmarked programs (Triple, Queens, Count, and Generator) can be expressed in both λ_{Cap} and $\mathbb{\lambda}_{\mathsf{Cap}}$. All except for Generator are taken from the literature.

The results are shown in Figure 5.7. All benchmarks were executed on a 2.60GHz Intel(R) Core(TM) i7 with 11GB of RAM. We compare our implementations with Koka (0.9.0) [Leijen, 2017b], Multicore OCaml (4.06.1) [Dolan et al., 2014], and an implementation of delimited control operators in Chez Scheme (9.5.3) [Dybvig et al., 2007]. For each comparison, we generate code in CPS in the corresponding language (that is, JavaScript, OCaml, and Scheme) and make sure to use the same primitive functions and data structures that the baseline uses. For each of the example functions we generate code using the translations of λ_{Cap} (Figure 5.4) and $\mathbb{\lambda}_{\mathsf{Cap}}$ (Figure 5.6). In our implementation of both translations, we additionally apply standard techniques [Danvy and Filinski, 1992, Schuster and Brachthäuser, 2018] to avoid generating administrative eta redexes, although these are not shown in Figure 5.4. We report the mean and standard deviation of the runtime of the programs under consideration.

We report numbers for four example programs. The Triple program is inspired by the example in Danvy and Filinski [1990]. It uses the running example choice from Section 5.1 to find triples of numbers that sum up to a given target number. The Queens example places queens on a chess board and is taken from Kiselyov and Sivaramakrishnan [2018]. The Count benchmark appears in Kammar et al. [2013], Kiselyov and Ishii [2015], and Wu and Schrijvers [2015] and counts down recursively using a single state effect. The Generator benchmark uses an effect operation to yield numbers which are summed up by a calling function using a state effect.

We now discuss the setup and observations specific to each of the baselines we compare against.

Benchmark	Time in ms (Standard Deviation) Baseline	λ_{Cap}	$\boldsymbol{\lambda}_{\mathsf{Cap}}$	Native
Koka				
Triple	2504.1 ±19.5	66.2 ±2.2	23.9 ±0.6	6.2 ±0.2
Queens (18)	403.4 ±9.3	170.8 ±1.7	171.4 ±1.2	161.9 ±4.1
Count (2K)	56.0 ±1.8	0.4 ±0.0	0.2 ±0.0	0.0 ±0.0
Generator (1K)	43.9 ±1.8	0.4 ±0.0	0.1 ±0.0	0.0 ±0.0
Chez Scheme				
Triple	68.6 ±1.1	3.7 ±0.1	3.7 ±0.1	1.8 ±0.0
Queens (18)	93.7 ±3.5	89.6 ±0.6	88.1 ±1.0	89.5 ±1.2
Count (1M)	445.2 ±27.2	10.5 ±0.6	10.5 ±0.8	1.9 ±0.0
Generator (1M)	664.2 ±14.6	17.6 ±0.5	17.7 ±0.5	2.1 ±0.0
Multicore OCaml				
Triple	25.0 ±2.4	4.5 ±0.1	2.4 ±0.1	2.0 ±0.1
Queens (18)	57.9 ±2.2	33.1 ±0.7	33.7 ±0.6	34.8 ±2.7
Count (1M)	72.5 ±0.9	19.4 ±0.5	7.5 ±0.2	2.8 ±0.0
Generator (1M)	93.9 ±1.3	18.3 ±0.5	10.3 ±0.3	3.9 ±0.1
Primes (1K)	32.2 ±0.6	29.0 ±0.6	22.8 ±0.4	N/A
Chameneos	26.7 ±0.6	32.7 ±1.0	28.7 ±0.9	N/A

Figure 5.7: Comparing the performance of λ_{Cap} and $\boldsymbol{\lambda}_{\mathsf{Cap}}$ with Koka, Multicore OCaml, and Chez Scheme.

Comparison with Koka Koka compiles to JavaScript and uses a standard library of builtin functions and data types also compiled to JavaScript. In our comparison with Koka, we do not generate Koka but JavaScript code in CPS and use the same compiled standard library. Benchmarks were executed using the JavaScript library `benchmark.js`[3] on `node.js`[4] version 12.11.1. For the Count and Generator benchmarks, we had to use a smaller initial state than in the other comparisons because the code generated by Koka as well as the code generated by our translation leads to a stack overflow for larger numbers. Koka already performs a selective CPS transformation. However, removing the runtime search for handler implementations causes significant speedups.

Comparison with Multicore OCaml Multicore [Dolan et al., 2014] is a fork of the OCaml compiler [Leroy et al., 2017] that adds support for effect handlers. We compile the Multicore OCaml programs with the multicore variant and our generated code with the standard variant of the `ocamlopt` compiler (4.06.1). Each program is compiled to a standalone executable, and we measure the running time with the `bench` program[5]. In our comparison with Multicore OCaml, we benchmarked two additional examples

[3]https://benchmarkjs.com/
[4]https://nodejs.org
[5]http://hackage.haskell.org/package/bench

from an online repository of Multicore OCaml examples[6]: Chameneos and Primes. These two benchmarks exercise the use-case that Multicore OCaml was designed for, that is, resuming continuations only once. Our translation always supports resuming continuations multiple times, but still offers competitive performance. The two additional examples use native side effects like for example mutating a global queue which we make execute in the right order by inserting let bindings as part of our translation.

Comparison with Monadic Delimited Control on Chez Scheme We also assess the performance of our generated code relative to a fast implementation of delimited continuations without any effect handling code. For this comparison, we implemented the examples using ordinary functions that capture the current continuation via `shift0` [Danvy and Filinski, 1989]. We use the library described by [Dybvig et al., 2007] and compile the example programs as well as our generated code with the Chez Scheme compiler [Dybvig, 2006]. In all four benchmarks we do not observe any speedup of the code generated from the translation of λ_{Cap} over the code generated from the translation of $\boldsymbol{\lambda}_{\mathsf{Cap}}$ where we eliminate redexes during translation. We have investigated Chez Scheme's intermediate representation and confirmed that, after optimization, the code is indeed the same for λ_{Cap} and $\boldsymbol{\lambda}_{\mathsf{Cap}}$, except that sometimes a subexpression is let bound. Does this make the restriction of $\boldsymbol{\lambda}_{\mathsf{Cap}}$ and its translation in Figure 5.6 unnecessary? No, on the contrary: the type system of $\boldsymbol{\lambda}_{\mathsf{Cap}}$ is an important conceptual tool that guided us to a well-performing implementation where optimal compilation is guaranteed. The fact that Chez Scheme can optimize the program similar to our improved translation can be seen as additional practical evidence for Theorem 24.

Benchmark results The benchmark results are generally encouraging. They indicate that the code we generate for $\boldsymbol{\lambda}_{\mathsf{Cap}}$ (and λ_{Cap}) is significantly faster than the languages we compare against. For the Triple benchmark, we can observe speedups of 105x (λ_{Cap} 38x) compared to Koka, 11x (λ_{Cap} 6x) compared to Multicore OCaml, and 19x for both implementations compared to Chez Scheme. For the Count benchmark we observe speedups of 297x (λ_{Cap} 150x) compared to Koka, 10x (λ_{Cap} 4x) compared to Multicore OCaml, and 42x (λ_{Cap} 43x) compared to Chez Scheme. For the Generator benchmark we observe speedups of 409x (λ_{Cap} 118x) compared to Koka, 9x (λ_{Cap} 5x) compared to Multicore OCaml, and 38x (λ_{Cap} 38x) compared to Chez Scheme. All three benchmarks extensively use control effects and yield optimization opportunities across effect operations for us to exploit. In the other benchmarks, we observe some speedups as well. Interestingly, in the Queens benchmark we do not observe any speedup between our unstaged translation and our staged translation. It uses one effect operation in a single place and handles it as a loop, which our staged translation immediately residualizes.

Comparison with hand-written code using native effects As another point of reference, we have implemented the benchmark examples and manually restructured the

[6]https://github.com/kayceesrk/effects-examples

programs to avoid effect handlers and use native effects instead. For example, we used native mutable state instead of effect handlers. We were careful to keep the same number of library calls (for example `append` on lists). The results are listed in column "Native" in Figure 5.7. In the benchmarks for Koka, where we compare with the generated code in JavaScript, the time for the native benchmarks is below 0.05 and thus displayed as 0.0. The results indicate that there is still an order of magnitude difference between the code we generate as the translation of $\boldsymbol{\lambda}_{\mathsf{Cap}}$ and the hand-written code using native effects.

5.5 Related Work

In this chapter we combined explicit capability passing with an implementation of control effects by iterated CPS transformation and specialization of programs with regard to evidence terms. This combination allows us to exploit static information and enables compile-time optimizations of lexical effect handlers. Applying some restrictions in $\boldsymbol{\lambda}_{\mathsf{Cap}}$, we are able to guarantee that all overhead introduced by the handler abstraction is eliminated. In this section, we relate our compilation technique to existing work on compile-time optimization of effect handlers.

Kammar et al. [2013] present multiple translations of effect handlers into Haskell. They translate handler implementations to type class instances, turning handlers into dictionary parameters of effectful functions. This can be seen as some form of capability passing. Furthermore, they present a translation that uses nested applications of the continuation monad for multiple handlers. This translation is very similar to the translation of λ_{Cap} to iterated CPS that we present here. However, they rely on GHC to optimize the abstractions they introduce. They do not explicitly state their assumptions for efficient code generation, while with $\boldsymbol{\lambda}_{\mathsf{Cap}}$ we make such assumptions explicit.

Wu and Schrijvers [2015] consider effectful programs as a free monad over a signature of effect operations. They fuse multiple handlers to avoid building and then folding any intermediate free monad structure in memory. They achieve excellent performance on a number of benchmarks, which validates their optimization method. Their optimization crucially relies on inlining and function specialization. Since their implementation uses Haskell and GHC, they use Haskell type classes to trigger function specialization, but do not state the conditions for when this may or may not happen. To get access to the current continuation, they use nested layers of the codensity monad which is operationally the same as the continuation monad. This is a similarity to our translation to nested layers of CPS.

Karachalias et al. [2021] present a compile-time optimization technique for dynamic effect handlers. They use the explicit effect coercions inferred by [Saleh et al., 2018]. While their approach is to apply semantics preserving rewrite rules, we translate effect handlers to a 2-level lambda calculus in CPS and apply beta-reductions at compile time. Our approach has the advantage that our optimizations are semantics-preserving by construction. As a downside, our translation might miss optimization opportunities that are specific to effect handlers and only become apparent in a language where they are explicitly represented. They compare their approach to the compilation technique presented in this thesis [Karachalias et al., 2021] in Table 2. Our technique is still competitive, sometimes even with hand-written code.

5.6 Conclusion

In this chapter we have evaluated our compilation technique. We have presented λ_{Cap}, and a second language $\mathbb{\lambda}_{\mathsf{Cap}}$, whose type system restricts programs to make it possible to always statically know handler implementations. We have given a translation of λ_{Cap} to STLC that generates fast code. The translation of $\mathbb{\lambda}_{\mathsf{Cap}}$ exploits the explicitness of our compilation technique to eliminate all overhead introduced by abstracting over effect operations. The crucial ingredients are explicit capability passing, iterated continuation passing, and explicit subregion evidence.

However, we see potential for improvement in the future. We generate code in CPS, which can be disadvantageous for some languages or virtual machines. Targeting a language with support for the delimited control operator **shift0**, we could instead use **shift0** directly instead of translating to iterated CPS. However, implementing delimited control in terms of iterated CPS is crucial to achieve compile-time optimization. It allows us to make use of the static knowledge of the context around the invocation of effect operations. In the future we want to investigate a direct-style translation that transforms programs in iterated CPS back to direct style, inserting the control operator **shift0** only where necessary.

It is common practice to compile to CPS [Kennedy, 2007], an explicit representation of join points [Maurer et al., 2017], or both [Cong et al., 2019]. In the future, we want to target an intermediate language with an explicit representation of continuations and treat continuations differently from functions at compile time and run time. For example, we could extend the intermediate language presented in [Kennedy, 2007] generalizing from two continuations to an arbitrary number.

Implementing programming languages involves a series of tradeoffs, usually on a spectrum between dynamic and static. This is no different for the implementation of effect handlers. By using explicit capability passing, iterated continuation passing, and monomorphizing effect-polymorphic function, we have explored the static end of this spectrum in detail and offer two data points in the design space of effect handlers. Other tradeoffs are possible. More experimentation with different designs and implementations of languages with effect handlers will help to inform these.

6 Conclusion

In this thesis we presented a compilation technique for lexical effect handlers. The key ideas are capability passing, continuation passing, and evidence passing. To summarize, the compilation technique presented in this thesis has the following advantages:

– It does not require a runtime system. It can target any language that supports first class functions, making it widely deployable.

– It does not reify computation as a value of a recursive data type. Generated programs are completely tagless.

– It produces programs in continuation-passing style, a well-studied intermediate representation in compilers. Generated programs are easily optimizable using well-known techniques.

– It produces well-typed programs in System F, a well-studied language. We do not introduce any sources of non-termination into the target language.

– In this thesis we prove multiple theorems of correctness about it. Most of these proofs are straight-forward, as we are careful to make them hold by construction.

It has the following disadvantages:

– It produces programs in continuation-passing style, which allocates stack frames on the heap.

– It increases the number of parameters of functions and makes extensive use of currying which might confuse arity analysis in existing compilers.

– It works for lexical effect handlers only which are less powerful than dynamic effect handlers.

– It is not clear if and how it is possible to support shallow effect handlers.

At the time of writing, we are investigating several improvements, extensions, and applications of the compilation technique presented in this thesis. Indeed, there is a lot to do.

Bibliography

Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321635361.

Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the servo web browser engine using rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 81–89, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342056. doi: 10.1145/2889160.2889229. URL https://doi.org/10.1145/2889160.2889229.

Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures*, pages 117–309. Oxford University Press, New York, NY, USA, 1992.

Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 1–16, Berlin, Heidelberg, 2013. Springer.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. doi: 10.1016/j.jlamp.2014.02.001.

Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly typed term representations in coq. *Journal of automated reasoning*, 49(2):141–159, 2012.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:The Calculus of Inductive Constructions*. Springer-Verlag, 2004.

Malgorzata Biernacka, Dariusz Biernacki, and Sergueï Lenglet. Typing control operators in the cps hierarchy. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, page 149–160, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307765. doi: 10.1145/2003476.2003498. URL https://doi.org/10.1145/2003476.2003498.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL):8:1–8:30, December 2017. ISSN 2475-1421. doi: 10.1145/3158096.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proc. ACM Program. Lang.*, 3(POPL):6:1–6:28, January 2019a. ISSN 2475-1421.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: Effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL), December 2019b. doi: 10.1145/3371116.

Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: Extensible algebraic effects in Scala (short paper). In *Proceedings of the International Symposium on Scala*, New York, NY, USA, 2017. ACM. doi: 10.1145/3136000.3136007.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effect handlers for the masses. *Proc. ACM Program. Lang.*, 2(OOPSLA):111:1–111:27, October 2018. ISSN 2475-1421. doi: 10.1145/3276481.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming*, 2020. doi: 10.1017/S0956796820000027.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. Versatile event correlation with algebraic effects. *Proc. ACM Program. Lang.*, 2(ICFP):67:1–67:31, July 2018a. ISSN 2475-1421.

Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. Versatile event correlation with algebraic effects. *Proc. ACM Program. Lang.*, 2(ICFP):67:1–67:31, July 2018b. ISSN 2475-1421.

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.*, 3(ICFP):79:1–79:28, July 2019. ISSN 2475-1421. doi: 10.1145/3341643.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *Journal of Functional Programming*, 30:e9, 2020. doi: 10.1017/S0956796820000039.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 315–326, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938152. doi: 10.1145/1291151.1291199. URL https://doi.org/10.1145/1291151.1291199.

Oliver Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. doi: 10.1017/S0960129500001535.

Olivier Danvy. On evaluation contexts, continuations, and the rest of computation. 02 2004.

Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. *DIKU Rapport 89/12, DIKU, University of Copenhagen*, 1989.

Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the Conference on LISP and Functional Programming*, pages 151–160, New York, NY, USA, 1990. ACM.

Stephen Dolan, Leo White, and Anil Madhavapeddy. Multicore OCaml. In *OCaml Workshop*, 2014.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, 2015.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *Proceedings of the Symposium on Trends in Functional Programming*. Springer LNCS 10788, 2017.

R. Kent Dybvig. The development of chez scheme. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 1–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: 10.1145/1159803.1159805. URL http://doi.acm.org/10.1145/1159803.1159805.

R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, November 2007. ISSN 0956-7968. doi: 10.1017/S0956796807006259.

Richard A. Eisenberg, Joachim Breitner, and Simon Peyton Jones. Type variables in patterns. In *Proceedings of the Haskell Symposium*, Haskell 2018, page 94–105, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3242744.3242753.

Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 180–190, New York, NY, USA, 1988. ACM.

Matthew Fluet and Greg Morrisett. Monadic regions. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, page 103–114, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581139055. doi: 10.1145/1016850.1016867.

James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Publishing Co., Boston, MA, USA, 1996. ISBN 0201634511.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 282–293, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134630. doi: 10.1145/512529.512563.

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 12–23, New York, NY, USA, 1995. ACM.

John Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, May 1998. doi: 10.1017/S0956796898003025.

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.

Daniel Hillerström, Sam Lindley, Bob Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In *Formal Structures for Computation and Deduction*, volume 84 of *LIPIcs*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, 30:e5, 2020. doi: 10.1017/S0956796820000040.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 1993.

Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In *International Workshop on Computer Science Logic*, pages 442–457. Springer, 2004.

Ohad Kammar and Matija Pretnar. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming*, 27(1), January 2017.

Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the International Conference on Functional Programming*, pages 145–158, New York, NY, USA, 2013. ACM.

Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. Efficient compilation of algebraic effect handlers. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi: 10.1145/3485479. URL https://doi.org/10.1145/3485479.

Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 177–190, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938152. doi: 10.1145/1291151.1291179.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the Haskell Symposium*, pages 94–105, New York, NY, USA, 2015. ACM. doi: 10.1145/2887747.2804319.

Oleg Kiselyov and Chung-chieh Shan. A substructural type system for delimited continuations. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 223–239, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73228-0. doi: 10.1007/978-3-540-73228-0_17. URL https://doi.org/10.1007/978-3-540-73228-0_17.

Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in OCaml. In *ML Workshop*, 2016.

Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in OCaml. In Kenichi Asai and Mark Shinwell, editors, *Proceedings of the ML Family Workshop / OCaml Users and Developers workshops*, volume 285 of *Electronic Proceedings in Theoretical Computer Science*, pages 23–58. Open Publishing Association, 2018. doi: 10.4204/EPTCS.285.2.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 285–299, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009880. URL https://doi.org/10.1145/3009837.3009880.

Daan Leijen. Extensible records with scoped labels. In *Proceedings of the Symposium on Trends in Functional Programming*, pages 297–312, 2005.

Daan Leijen. Koka: Programming with row polymorphic effect types. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, 2014.

Daan Leijen. Algebraic effects for functional programming. Technical report, MSR-TR-2016-29. Microsoft Research technical report, 2016.

Daan Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the Workshop on Type-Driven Development*, pages 16–29, New York, NY, USA, 2017a. ACM.

Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 486–499, New York, NY, USA, 2017b. ACM. doi: 10.1145/3093333.3009872.

Daan Leijen. First class dynamic effect handlers: Or, polymorphic heaps with dynamic effect handlers. In *Proceedings of the Workshop on Type-Driven Development*, pages 51–64, New York, NY, USA, 2018. ACM.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.06. *Institut National de Recherche en Informatique et en Automatique*, 2017.

Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003. ISSN 0890-5401. doi: 10.1016/S0890-5401(03)00088-9. URL `https://doi.org/10.1016/S0890-5401(03)00088-9`.

Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 500–514, New York, NY, USA, 2017. ACM. doi: 10.1145/3009837.3009897.

Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In *Proceedings of the International Conference on Functional Programming*, pages 81–93, New York, NY, USA, 2011. ACM.

Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, pages 296–311, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35182-2. doi: 10.1007/978-3-642-35182-2_21. URL `https://doi.org/10.1007/978-3-642-35182-2_21`.

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062380.

Flemming Nielson and Hanne Riis Nielson. Multi-level lambda-calculi: an algebraic description. In *Partial evaluation*, pages 338–354. Springer, 1996.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Type and effect systems. In *Principles of Program Analysis*, pages 283–363. Springer, 1999.

Martin Odersky. The Scala language specification, 2.8, 2019. URL `https://docs.scala-lang.org/tour/implicit-parameters.html`. [Last access: 09-30-2019].

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 234–251, New York, NY, USA, 2016. ACM. doi: 10.1145/3022671.2984009.

Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM. doi: http://doi.acm.org/10.1145/53990.54010.

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 809–818, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5583-4. doi: 10.1145/3209108.3209166. URL `http://doi.acm.org/10.1145/3209108.3209166`.

Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer-Verlag, 2009. doi: 10.1007/978-3-642-00590-9_7.

Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. doi: 10.2168/LMCS-9(4:23)2013.

Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM.

Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. Explicit effect subtyping. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 327–354, Cham, Switzerland, 2018. Springer International Publishing. ISBN 978-3-319-89884-1. doi: 10.1007/978-3-319-89884-1_12.

Philipp Schuster and Jonathan Immanuel Brachthäuser. Typing, representing, and abstracting control. In *Proceedings of the Workshop on Type-Driven Development*, pages 14–24, New York, NY, USA, 2018. ACM. doi: 10.1145/3240719.3241788.

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. Region-based resource management and lexical exception handlers in continuation-passing style. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 492–519, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99336-8. doi: 10.1007/978-3-030-99336-8_18.

Dorai Sitaram. Handling control. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 147–155, New York, NY, USA, 1993. ACM.

Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., USA, 3rd edition, 1997. ISBN 0201889544.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*, pages 203–217, New York, NY, USA, 1997. ACM.

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, October 2000. URL http://dx.doi.org/10.1016/S0304-3975(00)00053-0.

Hayo Thielecke. From control effects to typed continuation passing. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 139–149, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136285. doi: 10.1145/604131.604144.

Peter J. Thiemann. Cogen in six lines. In *Proceedings of the International Conference on Functional Programming*, pages 180–189, New York, NY, USA, 1996. ACM.

Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, February 1997. ISSN 0890-5401. doi: 10.1006/inco.1996.2613.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.

Nicolas Wu and Tom Schrijvers. Fusion for free - efficient algebraic effect handlers. In *Proceedings of the Conference on Mathematics of Program Construction*. Springer LNCS 9129, 2015.

Ningning Xie and Daan Leijen. Generalized evidence passing for effect handlers: Efficient compilation of effect handlers to c. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi: 10.1145/3473576. URL https://doi.org/10.1145/3473576.

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3408981.

Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL):5:1–5:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290318.

Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. Accepting blame for safe tunneled exceptions. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 281–295, New York, NY, USA, 2016. ACM.

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. Handling bidirectional control flow. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428207. URL https://doi.org/10.1145/3428207.