Typed Continuation-Passing for Lexical Handlers

Philipp Schuster
University of Tübingen, Germany

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Marius Müller
University of Tübingen, Germany

Klaus Ostermann
University of Tübingen, Germany

Abstract

Effect handlers are a language feature which enjoys popularity in academia and is also gaining traction in industry. Programs use abstract effect operations and handlers provide meaning to them in a delimited scope. Each effect operation is handled by the dynamically closest handler. Using an effect operation outside of a matching handler is meaningless and results in an error. A type-and-effect system prevents such errors from happening.

Lexical effect handlers are a recent variant of effect handlers with a number of attractive properties. Just as with traditional effect handlers, programs use effect operations and handlers give meaning to them. But unlike with traditional effect handlers, the connection between effect operations and their handler is lexical. Consequently, they typically have different type-and-effect systems.

The semantics of lexical effect handlers as well as their implementations use multi-prompt delimited control. They rely on the generation of fresh labels at runtime, which associate effect operations with their handlers. This use of labels and multi-prompt delimited control is theoretically and practically unsatisfactory.

Our main result is that typed lexical effect handlers do not need the full power of multi-prompt delimited control. We present the first CPS translation for lexical effect handlers to pure System F. It preserves well-typedness and simulates the traditional operational semantics. Importantly, it does so without requiring runtime labels. The CPS translation can be used to study the semantics of lexical effect handlers as well as as an implementation technique.

1 Introduction

Effect handlers [20, 21] are an attractive language feature, which subsumes a number of useful features, like exceptions, generators, asynchronous programming, etc. [22]. Recently, we have seen huge progress in both the theoretical foundations and the practical implementation of effect handlers [10, 16, 30, 27].

1.1 Lexical Effect Handlers

One such advancement are lexical effect handlers [6, 32, 2], a variant of effect handlers that enable lexical reasoning in higher-order programs. The semantics of lexical effect handlers differs from the traditional dynamic handlers in the higher-order case [2]. Where dynamic effect handlers always handle an effect operation with the dynamically closest handler for the effect, lexical effect handlers associate a unique label to each handler instance at runtime. Programs close over those labels, which leads to lexical scoping of effects. Lexical effect handlers are the basis for new forms of type- and effect systems [6] and equip programmers with new tools for reasoning [32, 2].

Effect safety for lexical effect handlers means that whenever an effect instances is used, the handler with the corresponding label is on the runtime stack. To guarantee effect safety, languages with lexical effect handlers typically have a type-and-effect system which tracks

Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. "Typed Continuation-Passing for Lexical Handlers". Technical Report. 2022. *University of Tübingen, Germany.*

the set of effect instances a program may use. Operationally, this set of effect instances corresponds to the set of labels that have to be on the runtime stack for a program to run.

1.2 The Problem

The semantics of lexical effect handlers based on fresh runtime labels directly leads to an implementation in terms of multi-prompt delimited control [11]. While this is certainly viable it is not fully satisfactory.

It is unsatisfactory from a *theoretical* point of view, since the semantic domain of multi-prompt delimited control by itself does not guarantee effect safety, as prompts can be undelimited. This makes external safety arguments necessary. Moreover, multi-prompt delimited control introduces non-termination into an otherwise total language.

It is unsatisfactory from a *practical* point of view, since implementations of multi-prompt delimited control either require runtime support [17], or a monadic translation that uses recursive data structures [11]. Moreover, they require a source of fresh labels and perform a recursive search for the correct label at runtime. This recursive search has a run-time cost and hinders compile-time optimizations.

Ideally, we would like to have a semantics for lexical effect handlers by translation into just System F, without any tags, labels, or recursion. This way, lexical handlers can be understood in terms of a well-studied, precise, and inherently safe semantic domain. Moreover, programs translated to System F do not require any runtime support and immediately lend themselves to optimizations by ordinary beta reduction. But is this possible?

1.3 Our Solution

Luckily, it turns out that typed lexical effect handlers do not need the full power of multi-prompt delimited control. In this paper, 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.

As a first step, we make a trivial observation: Lexical handlers are lexical. This suggests that we can rely on existing research on controlling lexical resources, such as regions [28, 13]. This observation is not new and appears for example in [32] and [2]. In this paper, we take the leap and talk about regions rather than effects because it feels more natural. It is important to note that by "region" we mean the abstract concept of a lexical scope and do not refer to a memory management technique.

Moreover, we follow Fluet and Morrisett [12] in their Single Effect Calculus, and express effectful function types like

$$\tau \to [\varepsilon_1, \varepsilon_2, ..., \varepsilon_n]\ \tau'$$

as

$$\forall [r; r \sqsubseteq \varepsilon_1, r \sqsubseteq \varepsilon_2, ..., r \sqsubseteq \varepsilon_n]\ \tau \to r\ \tau'$$

replacing sets of effects ε_1, ..., ε_n by a single *region* (*e.g.*, r) together with *subregioning evidence* (*e.g.*, $r \sqsubseteq \varepsilon_i$).

Our CPS translation now rests on two key ideas. Firstly, instead of passing freshly generated runtime labels and allocating the handler implementation on the runtime stack [2],

we directly pass the implementation of the corresponding effect handler [32, 6, 26]. Secondly, to find the correct delimiter, we interpret subregioning 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. In a polymorphic, higher-order language with control effects it is not immediately clear that this is always the case. Our contribution is to show that, for typed lexical effect handlers, it is.

1.4 Outline and Contributions

In Section 2, we informally present Λ_{cap}, a calculus with lexical effect handlers and a type-and-effect system that guarantees effect safety. We motivate the key difficulties and explain our solution by using concrete examples.

In Section 3, we formally present Λ_{cap}. We define the standard operational semantics as an abstract machine that generates a fresh label for each handler instance at runtime. We prove Progress and Preservation for this abstract machine (Theorems 4 and 5). Effect safety follows as a simple corollary. This result is not novel, but the proof itself is, since it directly uses the syntactic method [29].

In Section 4, we present a CPS translation from Λ_{cap} to pure System F. The translation takes well-typed programs in Λ_{cap} to well-typed terms in System F (Theorem 8). Since well-typed programs in System F never get stuck, this entails effect safety of Λ_{cap} in yet another novel way. We then prove that the CPS translation simulates the operational semantics (Theorem 10). This theorem is our main contribution. It is surprising, since the operational semantics uses labels to find handlers on the stack, while the CPS translation targets pure System F, without any labels, mutable state, or recursive types.

In Section 5 we compare to related work and in Section 6 we conclude.

2 Overview

In this section, we give an overview over our source language Λ_{cap} and the key ideas in this paper. Λ_{cap} is not meant to be used directly. Its purpose is similar to ExEff [24] which makes effects and subeffecting very explicit. Since Λ_{cap} is not designed to be user facing, we introduce it by comparison with examples written in Helium [2]. While we use the language Helium for illustration, other languages with lexical effect handlers such as Olaf [32, 34] or Scala Effekt [5] have very similar type systems and operational semantics. Λ_{cap} is a mostly standard call-by-value functional language with multi-arity functions. It has one additional feature: lexical effect handlers. Its type-system is also mostly standard, but features a region system with subregioning and explicit subregion evidence.

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 effect instance is used in the dynamic extent of the corresponding handler. We reuse existing work and call this dynamic extent a *region*. To guarantee effect safety, we keep track of the regions in which a computation can safely run.

2.1 Using Lexical Effects

As a starting point, consider the following simple example in Helium, which asks for two numbers and adds them:

```
signature Ask =
| ask : Unit ⇒ Int

let askTwice ‘a1 ‘a2 () = ask ‘a1 () + ask ‘a2 ()
```

We explicitly abstract over and pass two different effect instances `‘a1` and `‘a2` of the same effect `Ask`. The function `askTwice` has the following type:

```
(‘a1 : Ask) → (‘a2 : Ask) → Unit →[‘a1,‘a2] Int
```

It receives two instances of the effect `Ask` that appear in the effect `[‘a1, ‘a2]` of the function. This effect does not merely tell us that `askTwice` has some `Ask` effect, but that it uses these very effect instances.

▶ **Example 1.** Now consider the same function written in Λ_{cap}. We split the concept of effect instances (*e.g.*, `‘a1` and `‘a2`), which appear on the term level and on the type level, into term-level capabilities (*e.g.*, ask_1 and ask_2) and type-level regions (*e.g.*, r_1 and r_2).

$$
\begin{array}{l}
\textbf{def}\ \mathsf{askTwice}[\mathsf{r},\ \mathsf{r}_1,\ \mathsf{r}_2\ ;\ \mathsf{n}_1 :\ \mathsf{r} \sqsubseteq \mathsf{r}_1,\ \mathsf{n}_2 :\ \mathsf{r} \sqsubseteq \mathsf{r}_2](\\
\quad \mathsf{ask}_1 :\ \mathsf{Ask}[\mathsf{r}_1],\ \mathsf{ask}_2 :\ \mathsf{Ask}[\mathsf{r}_2]\\
)\ \textbf{at}\ \mathsf{r}\ \{\\
\quad \textbf{do}\ \mathsf{ask}_1[\mathsf{n}_1](\mathsf{unit})\ +\ \textbf{do}\ \mathsf{ask}_2[\mathsf{n}_2](\mathsf{unit})\\
\}
\end{array}
$$

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 [12], in Λ_{cap} there are no compound effects (such as `[‘a1, ‘a2]`). Instead, a function like $\mathsf{askTwice}$ 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 $\mathsf{askTwice}$ 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 has the same purpose as the corresponding type in Helium, while being more explicit.

2.2 Handling Lexical Effects

Lexical effect handlers introduce a name for each effect instance. Consider the following example in Helium:

```
handle ‘a1 in
  handle ‘a2 in
    askTwice ‘a1 ‘a2 ()
  with | ask () ⇒ resume 42 end
with | ask () ⇒ resume 43 end
```

In this example `'a1` and `'a2` are the names of two different instances of the `Ask` effect. We explicitly pass these effect instances to `askTwice`. Within `askTwice` there are two uses of the `ask` operation. Each of them will refer to a different handler. The example evaluates to `85`. It is possible to swap effect instances or to pass the same effect instance twice.

▶ **Example 2.** 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.*, ask_1) containing the handler implementation. The capability's region is the freshly introduced region.

$$
\begin{array}{l}
\textbf{try}\ \{\ [\mathsf{r}_1\ ;\ \mathsf{n}_1 :\ \mathsf{r}_1 \sqsubseteq \top](\mathsf{ask}_1 :\ \mathsf{Ask}[\mathsf{r}_1]) \Rightarrow \\
\quad \textbf{try}\ \{\ [\mathsf{r}_2\ ;\ \mathsf{n}_2 :\ \mathsf{r}_2 \sqsubseteq \mathsf{r}_1](\mathsf{ask}_2 :\ \mathsf{Ask}[\mathsf{r}_2]) \Rightarrow \\
\qquad \mathsf{askTwice}[\mathsf{r}_2,\ \mathsf{r}_1,\ \mathsf{r}_2\ ;\ \mathsf{n}_1,\ \mathbb{0}](\mathsf{ask}_1,\ \mathsf{ask}_2) \\
\quad \}\ \textbf{with}\ \{\ (\mathsf{u},\ \mathsf{k}) \Rightarrow \mathsf{k}(42)\ \} \\
\}\ \textbf{with}\ \{\ (\mathsf{u},\ \mathsf{k}) \Rightarrow \mathsf{k}(43)\ \}
\end{array}
$$

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 effect r_1, and the call to $\mathsf{askTwice}$ is checked in effect r_2. The evidence each handler introduces witnesses that the freshly introduced region subsumes the outer region.

When compared with Helium, in Λ_{cap} the call to $\mathsf{askTwice}$ is more explicit as 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.

2.3 Lexical Reasoning

Lexical handlers are not only useful to disambiguate different instances of the same effect [7]. They also offer improved reasoning tools in the presence of higher-order functions [33, 32]. Consider the following example, again in Helium:

```
handle 'exc1 in
  let abort () = fail 'exc1 () in
  handle 'exc2 in
    abort ()
  with | fail u ⇒ "aborted two" end
with | fail u ⇒ "aborted one" end
```

We install an exception handler and then define a function `abort`, which immediately fails. Because handlers in Helium are lexical, we know that the function `abort` will always abort to the handler which introduced the effect instance `'exc1`. This is the case, even if it is used under another exception handler and even if this handler was installed inside of another function. As we will see later, operationally, `abort` closes over a fresh label that is bound to `'exc1`. Hence the name "lexical" effect handler.

▶ **Example 3.** Again, the same example in Λ_{cap} is more explicit:

```
try { [r1, n1 : r1 ⊑ ⊤](exc1 : Exc[r1]) ⇒
  def abort[r, n : r ⊑ r1]() at r { do exc1[n](unit) };
  try { [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.

2.4 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.

2.4.1 Step One: Handler Passing

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

$$\langle\ \textbf{do cap}_{@3a1}\ \{\ (u,\ k) \Rightarrow k(42)\ \}[\bullet](unit)\ \ \|\ \ 43\ +\ \square\ ::\ \#_{@3a1}\ \{\ \square\ \}\ ::\ \#_{@b29}\ \{\ \square\ \}\ ::\ \bullet\ \rangle$$

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 $\textbf{cap}_{@3a1}\{\ (u,\ k)\ \Rightarrow\ k(42)\ \}$. The capability consists of the runtime label @3a1 as well as the handler implementation.

Similarly, our CPS translation does not rely on runtime labels to find the correct handler implementation. Again, we pass the handler implementation down to its use-site. The translation of this machine state is the following.

$$((\lambda u.\ \lambda k.\ k\ 42)\ unit)\ (\lambda x.\ (\lambda x_1.\ \lambda k_1.\ k_1\ x_1)\ (43\ +\ x))\ (\lambda x_2.\ \lambda k_2.\ k_2\ x_2)\ \texttt{done}$$

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. The label of the capability is associated with the first delimiter. The handler implementation will correctly capture the first of the three continuations.

2.4.2 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 subregioning evidence be lists of labels. This is important for our proof of effect safety (Theorem 6), 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 3, 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 $\mathsf{r}_2 \sqsubseteq \mathsf{r}_1$. After taking a few steps, the state of our abstract machine looks like this:

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

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 [31], we will show that this is always the case (Theorem 7). The central idea of this paper 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:

$$(\Lambda\mathsf{a}.\ \lambda\mathsf{m}.\ \lambda\mathsf{k}.\ \lambda\mathsf{j}.\ \mathsf{m}\ (\lambda\mathsf{x}.\ \mathsf{k}\ \mathsf{x}\ \mathsf{j}))\ ((\lambda\mathsf{u}.\ \lambda\mathsf{k}.\ \lambda\mathsf{k}_3.\ \mathsf{k}_3\ \texttt{"aborted one"})\ \mathsf{unit})\ (\lambda\mathsf{x}_1.\ \lambda\mathsf{k}_1.\ \mathsf{k}_1\ \mathsf{x}_1)\ (\lambda\mathsf{x}_2.\ \lambda\mathsf{k}_2.\ \mathsf{k}_2\ \mathsf{x}_2)\ \texttt{done}$$

The first two lines correspond to the statement that uses the capability. As before, the translated handler implementation (second line) is applied to the argument unit. Importantly, we translate the singleton evidence `@8ab` :: • to the function in the first line, called Lift [25]. 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}.

3 A Calculus with Lexical Effect Handlers

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 4) and preservation (Theorem 5). Effect safety then follows as a corollary: whenever we use an effect, the corresponding handler is on the stack (Corollary 6). Moreover, we establish the correspondence between type-level regions and term-level evidence (Corollary 7).

The paper is accompanied by a mechanized formalization of Λ_{cap} and its operational semantics in the Coq theorem prover [1], including the theorems of progress and preservation. The mechanization also includes the translation to System F as well as a proof of well-typedness

Terms:

Statements

s	::=	**val** $x\ =\ s;\ s$	sequencing of comp.
	|	**return** v	returning values
	|	$v[\overline{\rho}\ ;\ \overline{e}](\overline{v})$	calling functions
	|	**do** $v[e](v)$	performing effects
	|	**try** $\{\ [r\ ;\ n](c) \Rightarrow s\ \}$ **with** $\{\ (x,\ k) \Rightarrow s\ \}$	handling effects

Values

v	::=	$x,\ f,\ c,\ ...$	variables
	|	$()\ \mid 0\ \mid 1\ \mid ...\ \mid \mathsf{true}\ \mid ...$	primitives
	|	$\{\ [\overline{r}\ ;\ \overline{n:\ \gamma}](\overline{x:\ \tau})$ **at** $\rho \Rightarrow s\}$	closures

Evidence

e	::=	$n,\ ...$	evidence variables
	|	$\mathbb{0}$	reflexive evidence
	|	$e\ \oplus\ e$	transitive evidence

Types:

Types

τ	::=	$\mathsf{Int}\ \mid \mathsf{Bool}\ \mid ...$	primitives
	|	$\forall[\overline{r}\ ;\ \overline{\gamma}](\overline{\tau}) \to\!\rho\ \tau$	functions
	|	**Cap** $\rho\ \tau\ \tau$	capabilities

Regions

ρ	::=	r	region variable
	|	$\top$	top-level region

Constraints

γ	::=	$\rho\ \sqsubseteq\ \rho$	subregion

Environments:

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

Figure 1 Syntax of Λ_{cap}.

preservation (Theorem 8). The mechanized formalization is attached as supplementary material.

3.1 Syntax

Figure 1 defines the syntax of Λ_{cap}. We use fine-grain call-by-value [19] and syntactically distinguish 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 **do** $v_0[e](v)$, where v_0 is the capability, e is evidence, and v is the argument. We handle a statement with **try** $\{\ ...\ \}$ **with** $\{\ ...\ \}$. 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 Variables stand for 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. 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 subregioning relationships.

We define the following short-hand notation for named function definitions:

$$\begin{aligned}&\textbf{def}\ f[\overline{r}\ ;\ \overline{n : \gamma}](\overline{x : \tau})\ \textbf{at}\ \rho\ \{\ s_0\ \};\ s \quad \doteq \\ &\quad \textbf{val}\ f\ =\ \textbf{return}\ \{\ [\overline{r}\ ;\ \overline{n : \gamma}](\overline{x : \tau})\ \textbf{at}\ \rho \Rightarrow s_0\};\ s\end{aligned}$$

The list of region parameters scopes over the evidence parameters types, the parameter types, the return type, the annotated region ρ, and the body s_0 of the function.

3.2 Typing

Figure 2 defines the typing rules of Λ_{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 [23]) 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 $\overline{v}$. 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. The evidence e witnesses that the region ρ of the statement subsumes the region of the capability ρ'. When a capability is used in a different context, we require explicit evidence that it is safe to do so. Again, there is no implicit subtyping and no subsumption rule. In rule Try, 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

Statement Typing

$$\boxed{\begin{array}{c}\Gamma \mid \rho \vdash\ s\ :\ \tau \\ \uparrow\ \uparrow\quad \uparrow\quad \downarrow\end{array}}$$

$$\frac{\Gamma \mid \rho \vdash\ s_0\ :\ \tau_0 \qquad \Gamma,\ x_0 :\ \tau_0 \mid \rho \vdash\ s\ :\ \tau}{\Gamma \mid \rho \vdash\ \textsf{val}\ x_0\ =\ s_0;\ s\ :\ \tau}\ [\textsc{Val}]$$

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

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

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

Value Typing

$$\boxed{\begin{array}{c}\Gamma \vdash\ v\ :\ \tau \\ \uparrow\quad \uparrow\quad \downarrow\end{array}}$$

$$\frac{\Gamma(x)\ =\ \tau}{\Gamma \vdash\ x\ :\ \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash\ 1\ :\ \mathsf{Int}}\ [\textsc{Lit}]$$

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

Evidence Typing

$$\boxed{\begin{array}{c}\Gamma \vdash\ e\ :\ \gamma \\ \uparrow\quad \uparrow\quad \downarrow\end{array}}$$

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

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

Figure 2 Type system of Λ_{cap}.

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. Evidence parameters allow us to write region-polymorphic functions that are *constrained* to

Syntax of Labels:

$l \quad ::= \quad \texttt{@a5f} \mid \texttt{@4b2} \mid \ldots$

Additional Runtime Values:

$v \quad ::= \quad \ldots \mid \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \} \mid \mathbf{resume}(\mathsf{F} :: \mathsf{H})$

Syntax of Machine States:

$$\begin{array}{lll} \mathsf{M} ::= & \langle\, s \parallel \mathsf{K} \,\rangle & \text{executing} \\ \mid & \langle\, \mathbf{do}\ v[w](v) \parallel \mathsf{K} \parallel \mathsf{H} \,\rangle & \text{unwinding} \\ \mid & \langle\, \mathbf{resume}(\mathsf{F} :: \mathsf{H})(v) \parallel \mathsf{K} \,\rangle & \text{rewinding} \end{array}$$

Syntax of Stacks:

$\mathsf{K} ::= \bullet \mid \mathsf{F} :: \mathsf{K}$

Syntax of Resumptions:

$\mathsf{H} ::= \bullet \mid \mathsf{F} :: \mathsf{H}$

Syntax of Frames:

$$\begin{array}{lll} \mathsf{F} ::= & \mathbf{val}\ x = \square;\ s \\ \mid & \#_l \{ \square \} \end{array}$$

(a) Syntax of the abstract machine.

Machine Steps:

$$\begin{array}{lll}
\textit{(return)} & \langle\, \mathbf{return}\ v \parallel \mathbf{val}\ x = \square;\ s :: \mathsf{K} \,\rangle & \to \langle\, s[x \mapsto v] \parallel \mathsf{K} \,\rangle \\
\textit{(push)} & \langle\, \mathbf{val}\ x = s_0;\ s \parallel \mathsf{K} \,\rangle & \to \langle\, s_0 \parallel \mathbf{val}\ x = \square;\ s :: \mathsf{K} \,\rangle \\
\textit{(call)} & \langle\, \{\ [\overline{r}\ ;\ \overline{n : \gamma}](\overline{x : \tau})\ \mathbf{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{(try)} & \langle\, \mathbf{try}\ \{\ [r\ ;\ n](c) \Rightarrow s_0\ \}\ \mathbf{with}\ \{\ (x,\ k) \Rightarrow s\ \} \parallel \mathsf{K} \,\rangle \to & \\
& \langle\, s_0[r \mapsto *, n \mapsto *, c \mapsto \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}] \parallel \#_l \{ \square \} :: \mathsf{K} \,\rangle & \\
& \quad \text{where } l = \texttt{generateFresh}() & \\
\textit{(pop)} & \langle\, \mathbf{return}\ v \parallel \#_l \{ \square \} :: \mathsf{K} \,\rangle & \to \langle\, \mathbf{return}\ v \parallel \mathsf{K} \,\rangle \\
\textit{(perform)} & \langle\, \mathbf{do}\ \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}[*](v) \parallel \mathsf{K} \,\rangle & \to \langle\, \mathbf{do}\ \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}[*](v) \parallel \mathsf{K} \parallel \bullet \,\rangle \\
\textit{(unwind)} & \langle\, \mathbf{do}\ \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}[*](v) \parallel \mathbf{val}\ x = \square;\ s :: \mathsf{K} \parallel \mathsf{H} \,\rangle \to & \\
& \langle\, \mathbf{do}\ \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}[*](v) \parallel \mathsf{K} \parallel \mathbf{val}\ x = \square;\ s :: \mathsf{H} \,\rangle & \\
\textit{(forward)} & \langle\, \mathbf{do}\ \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}[*](v) \parallel \#_{l'} \{ \square \} :: \mathsf{K} \parallel \mathsf{H} \,\rangle \to & \\
& \langle\, \mathbf{do}\ \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}[*](v) \parallel \mathsf{K} \parallel \#_{l'} \{ \square \} :: \mathsf{H} \,\rangle & \\
& \quad \text{where } l \neq l' & \\
\textit{(handle)} & \langle\, \mathbf{do}\ \mathbf{cap}_l \{ (x,\ k) \Rightarrow s \}[*](v) \parallel \#_{l'} \{ \square \} :: \mathsf{K} \parallel \mathsf{H} \,\rangle \to & \\
& \langle\, s[x \mapsto v][k \mapsto \mathbf{resume}(\#_{l'} \{ \square \} :: \mathsf{H})] \parallel \mathsf{K} \,\rangle & \\
& \quad \text{where } l = l' & \\
\textit{(rewind)} & \langle\, \mathbf{resume}(\mathsf{F}_1 :: \mathsf{F}_2 :: \mathsf{H})(v) \parallel \mathsf{K} \,\rangle & \to \langle\, \mathbf{resume}(\mathsf{F}_2 :: \mathsf{H})(v) \parallel \mathsf{F}_1 :: \mathsf{K} \,\rangle \\
\textit{(resume-1)} & \langle\, \mathbf{resume}(\mathbf{val}\ x = \square;\ s :: \bullet)(v) \parallel \mathsf{K} \,\rangle & \to \langle\, s[x \mapsto v] \parallel \mathsf{K} \,\rangle \\
\textit{(resume-2)} & \langle\, \mathbf{resume}(\#_l \{ \square \} :: \bullet)(v) \parallel \mathsf{K} \,\rangle & \to \langle\, \mathbf{return}\ v \parallel \mathsf{K} \,\rangle
\end{array}$$

(b) Steps of the abstract machine.

Figure 3 Syntax and reduction rules of the abstract machine for Λ_{cap}.

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. We require the composition of evidence to be associative.

3.3 Operational Semantics

Figure 3 lists the syntax and reduction rules of the abstract machine semantics of Λ_{cap}.

Labels As is common for lexical effect handlers, the semantics of Λ_{cap} is given in terms of a machine for multi-prompt delimited control. As mentioned in the introduction, our operational semantics uses labels l, which are freshly generated at runtime. Later in the CPS translation (Section 4), 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 $\mathbf{cap}_l\ \{\ ...\ \}$ which consist of a label l and a handler implementation. Secondly, continuations $\mathbf{resume}(\#_l\ \{\ \Box\ \}\ ::\ \mathsf{H})$ which contain a resumption of the form $\#_l\ \{\ \Box\ \}\ ::\ \mathsf{H}$.

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 $\mathbf{val}\ x\ =\ \Box;\ s$ or a delimiter frame $\#_l\ \{\ \Box\ \}$ with a label l. The executing state has the form $\langle\ s\ \|\ \mathsf{K}\ \rangle$. 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 3 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 full stepping relation is provided in Appendix A, which is attached as supplementary material. 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 *(try)* 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 *(handle)* 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 $\#_l\ \{\ \Box\ \}$. Rule *(rewind)* repushes the resumption onto the stack and resumes execution by returning the argument v to the stack.

3.4 Soundness

Λ_{cap} satisfies the standard soundness properties.

▶ **Theorem 4** (Progress).
If $\vdash\ M\ ok$, *then either* M *is of the form* $\langle\ \mathbf{return}\ v\ \|\ \bullet\ \rangle$ *for some value* v, *or* $M \to M'$ *for some machine* M'.

Extended syntax of evidence and regions:

e	$::=$	$\ldots \mid w$	evidence value
ρ	$::=$	$\ldots \mid u$	runtime region

Evidence values and runtime regions:

w	$::=$	$\bullet \mid l :: w$	evidence values
u	$::=$	$\bullet \mid l :: u$	runtime regions

Runtime region of stack:

$$
\begin{array}{lll}
\mathcal{R}[\![\cdot]\!] & : & \mathsf{K} \to u \\
\mathcal{R}[\![\bullet]\!] & = & \bullet \\
\mathcal{R}[\![\textbf{val}\ x = \square;\ s :: \mathsf{K}]\!] & = & \mathcal{R}[\![\mathsf{K}]\!] \\
\mathcal{R}[\![\#_l \{ \square \} :: \mathsf{K}]\!] & = & l :: \mathcal{R}[\![\mathsf{K}]\!]
\end{array}
$$

Normalization of evidence:

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

Abstract machine, evidence value, and capability typing:

$$\frac{\emptyset \mid \mathcal{R}[\![\mathsf{K}]\!] \vdash s : \tau \qquad \vdash \mathsf{K} : \tau}{\vdash \langle\, s \parallel \mathsf{K} \,\rangle\ ok}\ [\textsc{Machine}] \qquad \frac{u_0 = w +\!\!+ u_1}{\emptyset \vdash w : u_0 \sqsubseteq u_1}\ [\textsc{Evidence}]$$

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

Figure 4 Proof invariants of the abstract machine.

▶ **Theorem 5** (Preservation)**.**
If $\vdash M\ ok$ *and* $M \to M'$ *then* $\vdash M'\ ok$*.*

In order to prove progress and preservation, we need to establish invariants, which are maintained by machine reduction. Figure 4 lists the most important concepts needed for our proofs. The full typing rules for the abstract machine are provided in Appendix A.

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 handler frames 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 (Appendix A), we substitute runtime regions for region variables and evidence values for evidence variables. For example, in step *(try)* we substitute $l \;::\; \mathcal{R}[\![\, \mathsf{K} \,]\!]$ for the region variable r and we substitute the singleton runtime region $l \;::\; \bullet$ 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.

3.5 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 6** (Effect Safety)**.**
If $\langle$ ***do*** $\mathbf{cap}_l$ $\{\ (x,\ k) \Rightarrow s\ \}[e](v) \parallel K \rangle$ *ok, then* $l \in \mathcal{R}[\![\, K \,]\!]$.

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. [31].

▶ **Corollary 7** (Evidence Correspondence)**.**
If $\langle$ ***do*** $v_0[e](v) \parallel K \rangle$ *ok, then* $\mathcal{R}[\![\, K \,]\!] = \mathcal{N}[\![\, e \,]\!] +\!\!+ (l \;::\; u)$ *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 Continuation-Passing Style Translation

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 of this paper, by translating Λ_{cap} into pure System F, we show that labels, recursive datatypes, or mutable state are not necessary to implement statically typed, lexical effect handlers.

Our CPS translation can serve as a compilation technique for languages with lexical effect handlers into any language that supports first-class functions, which makes it widely applicable. Moreover, it is a generalization of the translation presented by Schuster et al. [26], which has been shown to enable compile-time optimizations for significant performance improvements.

We translate into one particular variant of CPS, called *iterated CPS* [9, 25]. Every stack segment, delimited by a label, is represented by its own continuation argument. In other words, in iterated CPS, functions do not receive one but potentially multiple continuations.

Translation of Types:

$$
\begin{array}{lcl}
\mathcal{T}[\![\ \mathsf{Int}\ ]\!] & = & \mathsf{Int} \\
\mathcal{T}[\![\ \forall[\overline{r},\overline{\gamma}](\overline{\tau}) \to \rho\ \tau_0\ ]\!] & = & \overline{\forall r.}\ \overline{\mathcal{T}[\![\gamma]\!] \to} \ \overline{\mathcal{T}[\![\tau]\!] \to}\ \textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ \mathcal{T}[\![\tau_0]\!] \\
\mathcal{T}[\![\ \mathbf{Cap}\ \rho\ \tau_1\ \tau_2]\!] & = & \mathcal{T}[\![\tau_1]\!] \to \textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ \mathcal{T}[\![\tau_2]\!] \\
\mathcal{T}[\![\ r\ ]\!] & = & r \\
\mathcal{T}[\![\ \top]\!] & = & \mathsf{Void} \\
\mathcal{T}[\![\ \rho\ \sqsubseteq\ \rho'\ ]\!] & = & \forall a.\ \textsc{Cps}\ \mathcal{T}[\![\rho']\!]\ a \to \textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ a
\end{array}
$$

Translation of Values:

$$
\begin{array}{lcl}
\mathcal{V}[\![\ x\ ]\!] & = & x \\
\mathcal{V}[\![\ 1\ ]\!] & = & 1 \\
\mathcal{V}[\![\ \{\ [\overline{r},\overline{n\ :\ \gamma}](\overline{x\ :\ \tau})\ \mathbf{at}\ \rho \Rightarrow s\ \}\ ]\!] & = & \overline{\Lambda r.}\ \overline{\lambda n.}\ \overline{\lambda x.}\ \mathcal{S}[\![s]\!]
\end{array}
$$

Translation of Evidence:

$$
\begin{array}{lcl}
\mathcal{E}[\![\ n\ ]\!] & = & n \\
\mathcal{E}[\![\ \mathbb{0}\ ]\!] & = & \Lambda a.\ \lambda m.\ m \\
\mathcal{E}[\![\ e_1\ \oplus e_2\ ]\!] & = & \Lambda a.\ \lambda m.\ \mathcal{E}[\![e_1]\!]\ a\ (\mathcal{E}[\![e_2]\!]\ a\ m)
\end{array}
$$

Translation of Statements:

$$
\begin{array}{lcl}
\mathcal{S}[\![\ \mathbf{return}\ v\ ]\!] & = & \lambda k.\ k\ (\mathcal{V}[\![v]\!]) \\
\mathcal{S}[\![\ \mathbf{val}\ x\ =\ s_0;\ s\ ]\!] & = & \lambda k.\ \mathcal{S}[\![s_0]\!]\ (\lambda x.\ \mathcal{S}[\![s]\!]\ k) \\
\mathcal{S}[\![\ v_0[\overline{\rho},\overline{e}](\overline{v})\ ]\!] & = & \mathcal{V}[\![v_0]\!]\quad \overline{\mathcal{T}[\![\rho]\!]}\quad \overline{\mathcal{E}[\![e]\!]}\quad \overline{\mathcal{V}[\![v]\!]} \\
\mathcal{S}[\![\ \mathbf{do}\ v_0[e](v)\ ]\!] & = & \mathcal{E}[\![e]\!]\quad \mathcal{T}[\![\tau_2]\!]\quad (\mathcal{V}[\![v_0]\!]\ \mathcal{V}[\![v]\!])
\end{array}
$$

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

Auxiliary Definitions:

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

Figure 5 Translation from Λ_{cap} to System F.

Figure 5 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. Capabilities are translated to functions in CPS.

Regions become answer types. The iterated CPS translation is very similar to the traditional CPS translation. In particular, the auxiliary meta-definition $\textsc{Cps}\ R\ A$ is defined as the familiar type $(A \to R) \to R$ of computations in CPS with *return type* A and *answer type* R.

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.

As usual in CPS, 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 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 Lift, and the capability $\lambda x.\ \lambda k.\ \mathcal{S}[\![s]\!]$. We use the meta function Reset to apply the whole term to an empty continuation argument.

4.1 Typability Preservation

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

▶ **Theorem 8** (Well-typedness of Translated Terms)**.**
If $\Gamma \shortmid \rho \vdash s : \tau$, *then* $\mathcal{T}[\![\ \Gamma\ ]\!] \vdash \mathcal{S}[\![\ s\ ]\!] : \textit{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \mathcal{T}[\![\ \tau\ ]\!]$

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 8 in the Coq proof assistant.

▶ **Example 9.** 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.

```
try { [r ; n](exc) ⇒ do exc[𝕆](unit) }
  with { (x, k) ⇒ return 1 }
```

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 Cps Void Int.

```
(Λr. λn. λexc. (Λa. λm. m) Int (exc unit))
  (Cps Void Int)  (Lift)  (λx. λk. λk₂. k₂ 1)
  (λx. λk. k x)
```

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, **do** exc[0](unit)) is typed in region r and consequently is translated to a term in CPS of type `Cps r Int`. We instantiate the polymorphic answer type r with the type `Cps Void Int`, which results in the overall type `Cps (Cps Void Int) 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 $\mathsf{Unit} \to \texttt{Cps}\ (\texttt{Cps}\ \mathsf{Void}\ \mathsf{Int})\ \mathsf{Int}$. It discards the first continuation and returns to the second.

4.2 Simulation

In Section 3, we defined an operational semantics of Λ_{cap}. In this section, we defined a CPS translation to System F in CPS. 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 10** (Simulation)**.**
If $\vdash M\ ok$ *and* $M \to M'$*, then* $\mathcal{M}[\![\ M\ ]\!] \to^* \mathcal{M}[\![\ M'\ ]\!]$.

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 [8]. 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 full translation is given in Appendix A.

The following corollary follows from Theorem 10. 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 11** (Evaluation)**.**
If $\emptyset \mid \top \vdash s : \mathit{Int}$ *and* $\langle\ s \mid\mid \bullet\ \rangle \to^* \langle\ \mathbf{return}\ v \mid\mid \bullet\ \rangle$
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.

▶ **Example 12.** Figure 6 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

Steps in the source language:

(1) ⟨ **do** $\mathsf{cap}_{@b29}\{\ (u,\ k_1) \Rightarrow k_1(43)\ \}[@3a1\ ::\ \bullet](\mathsf{unit})$ ‖ $\#_{@3a1}\ \{\ \square\ \}$ :: $\#_{@b29}\ \{\ \square\ \}$ :: $\bullet$ ⟩ →

(2) ⟨ **do** $\mathsf{cap}_{@b29}\{\ (u,\ k_1) \Rightarrow k_1(43)\ \}[@3a1\ ::\ \bullet](\mathsf{unit})$ ‖ $\#_{@3a1}\ \{\ \square\ \}$ :: $\#_{@b29}\ \{\ \square\ \}$ :: $\bullet$ ‖ $\bullet$ ⟩ →

(3) ⟨ **do** $\mathsf{cap}_{@b29}\{\ (u,\ k_1) \Rightarrow k_1(43)\ \}[\bullet](\mathsf{unit})$ ‖ $\#_{@b29}\ \{\ \square\ \}$ :: $\bullet$ ‖ $\#_{@3a1}\ \{\ \square\ \}$ :: $\bullet$ ⟩ →

(4) ⟨ **resume**$(\#_{@b29}\ \{\ \square\ \}$ :: $\#_{@3a1}\ \{\ \square\ \}$ :: $\bullet)(43)$ ‖ $\bullet$ ⟩ →

(5) ⟨ **resume**$(\#_{@3a1}\ \{\ \square\ \}$ :: $\bullet)(43)$ ‖ $\#_{@b29}\ \{\ \square\ \}$ :: $\bullet$ ⟩ →

(6) ⟨ **return** 43 ‖ $\#_{@b29}\ \{\ \square\ \}$ :: $\bullet$ ⟩ →

(7) ⟨ **return** 43 ‖ $\bullet$ ⟩

Steps in the target language:

(1) (Lift ((λu. λk_1. λk_2. k_1 43 k_2) unit)) (λx. λk. k x) (λx. λk. k x) done →

(2) (λk. λj. (λk_1. λk_2. k_1 43 k_2) (λy. k y j)) (λx. λk. k x) (λx. λk. k x) done →

(3) (λk_1. λk_2. k_1 43 k_2) (λy. (λx. λk. k x) y (λx. λk. k x)) done →

(4) (λy. (λx. λk. k x) y (λx. λk. k x)) 43 done →

(5) (λx. λk. k x) 43 (λx. λk. k x) done →

(6) (λk. k 43) (λx. λk. k x) done →

(7) (λk. k 43) done

Figure 6 Example of step-by-step simulation. The statement under reduction is highlighted in *grey*, the first stack segment highlighted in *blue*, the second stack segment highlighted in *yellow*, and the top of the stack highlighted in *red*.

(highlighted in grey), the context, and the trace are translated, and how the translated term captures the correct number of continuations and reinstalls them.

The first steps are *(perform)*, *(forward)*, and *(handle)*. The evidence value `@3a1 :: •` is precisely the offset we have to skip to get to the correct delimiter (Theorem 7). 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.

5 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 dynamic effect handlers.

5.1 Lexical Effect Handlers

There are a number of implementations of lexical effect handlers which do not guarantee effect safety. For example for OCaml [18], Scala [3], and Java [4]. All of these use multi-prompt delimited control [11] under the hood, therefore their operational semantics is comparable to our operational semantics given in Section 3. 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 prompt 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 [32, 5, 2]. All of these define effects as sets of effect instances a computation might use. In our type system in Section 3 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 [12], 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. [2], which uses reduction under binders.

Two languages with lexical effect handlers which guarantee effect safety but do not track sets of effect instances are the ones by Brachthäuser et al. [6] and by Schuster et al. [26]. Brachthäuser et al. have a multi-prompt semantics and rely on a second-class restriction to guarantee effect safety. Schuster et al. use lists of answer types as effects. Their CPS translation is very similar to ours, but their approach does not support effect polymorphism and they do not show any operational correspondence.

5.2 Dynamic Effect Handlers

Hillerström et al. [2017, 2020] present a CPS translation for dynamic effect handlers. They present an abstract machine and a simulation result. Their abstract machine searches for a matching effect tag on the stack, and so do their CPS translated terms. In contrast, we present a CPS translation for lexical effect handlers. Furthermore, our abstract machine searches for a matching label, but our CPS translated terms do not. Our CPS translation is similar to their curried CPS translation. They produce untyped terms, where our CPS translation produces typed System F terms. They sketch how a typed CPS translation might look like in Appendix B of [14], but do not present a fully worked proof of well-typedness, which we do.

Saleh et al. [24] 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 [16]. 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 subregioning evidence down to where effect operations are used and never coerce statements directly. Based on results by Schuster et al. [26], we conjecture that our iterated CPS translation produces efficient programs.

Xie et al. [31] and Xie and Leijen [30] present an implementation of dynamic effect handlers where evidence vectors are explicitly passed. Our formal treatment of the operational semantics is inspired by their concept of evidence correspondence. Furthermore, Xie et al. [31] introduce the property of scoped resumptions and dynamically check that it holds. In contrast, we statically enforce this property.

6 Conclusion

We have presented a CPS translation for a typed language with lexical effect handlers. It simulates the standard operational semantics, while targeting pure System F with neither labels nor runtime constructs. Our translation is based on two key ingredients: firstly, passing handler implementations instead of looking them up in the context. Secondly, interpreting lexical effects as regions and making subregioning explicit. 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 [34] could hold some interesting challenges.

References

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

2 D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Binders by day, labels by night: Effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019. doi: 10.1145/3371116.

3 J. I. Brachthäuser and P. 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.

4 J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effect handlers for the masses. *Proc. ACM Program. Lang.*, 2(OOPSLA):111:1–111:27, Oct. 2018. ISSN 2475-1421. doi: 10.1145/3276481.

5 J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming*, 2020. doi: 10.1017/S0956796820000027.

6 J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020. doi: 10.1145/3428194.

7 O. Bračevac, N. Amin, G. Salvaneschi, S. Erdweg, P. Eugster, and M. Mezini. Versatile event correlation with algebraic effects. *Proc. ACM Program. Lang.*, 2(ICFP):67:1–67:31, July 2018. ISSN 2475-1421.

8 O. Danvy. On evaluation contexts, continuations, and the rest of computation. 02 2004.

9 O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the Conference on LISP and Functional Programming*, pages 151–160, New York, NY, USA, 1990. ACM.

10 P. E. de Vilhena and F. Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434314. URL `https://doi.org/10.1145/3434314`.

11 R. K. Dybvig, S. L. Peyton Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.

12 M. Fluet and G. 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.

13 D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. 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.

14 D. Hillerström, S. Lindley, B. Atkey, and K. 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.

15 D. Hillerström, S. Lindley, and R. Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, 30:e5, 2020. doi: 10.1017/S0956796820000040.

16 G. Karachalias, F. Koprivec, M. Pretnar, and T. 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`.

17 O. Kiselyov. Delimited control in OCaml, abstractly and concretely. *Theoretical Computer Science*, 435:56–76, 2012.

18 O. Kiselyov and K. Sivaramakrishnan. Eff directly in OCaml. In *ML Workshop*, 2016.

19 P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

20 G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer-Verlag, 2009.

21 G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

22 M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015.

23 J. 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.

24 A. H. Saleh, G. Karachalias, M. Pretnar, and T. Schrijvers. Explicit effect subtyping. In A. Ahmed, editor, *Programming Languages and Systems*, pages 327–354, Cham, Switzerland, 2018. Springer International Publishing. ISBN 978-3-319-89884-1.

25 P. Schuster and J. I. 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.

26 P. Schuster, J. I. Brachthäuser, and K. Ostermann. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.*, 4(ICFP), Aug. 2020. doi: 10.1145/3408975.

27 K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy. Retrofitting effect handlers onto ocaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 206–221, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454039. URL `https://doi.org/10.1145/3453483.3454039`.

28 M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2): 109–176, Feb. 1997. ISSN 0890-5401. doi: 10.1006/inco.1996.2613.

29 A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994.

30 N. Xie and D. 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`.

31 N. Xie, J. I. Brachthäuser, D. Hillerström, P. Schuster, and D. Leijen. Effect handlers, evidently. *Proc. ACM Program. Lang.*, 4(ICFP), Aug. 2020. doi: 10.1145/3408981.

32 Y. Zhang and A. C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL):5:1–5:29, Jan. 2019. ISSN 2475-1421.

33 Y. Zhang, G. Salvaneschi, Q. Beightol, B. Liskov, and A. 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.

34 Y. Zhang, G. Salvaneschi, and A. C. Myers. Handling bidirectional control flow. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020. doi: 10.1145/3428207. URL `https://doi.org/10.1145/3428207`.

Abstract Machine Typing

$$\frac{\emptyset \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash\ s\ :\ \tau \qquad \vdash\ \mathsf{K}\ :\ \tau}{\vdash\ \langle\ s\ \|\ \mathsf{K}\ \rangle\ ok}\ [\textsc{Machine}] \qquad \frac{\begin{array}{l}\emptyset \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash\ \mathbf{do}\ h[w](v)\ :\ \tau_2 \\ \vdash\ \mathsf{K}\ :\ \tau \qquad \tau_2 \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash\ \mathsf{H}\ :\ \tau\end{array}}{\vdash\ \langle\ \mathbf{do}\ h[w](v)\ \|\ \mathsf{K}\ \|\ \mathsf{H}\ \rangle\ ok}\ [\textsc{Unwinding}]$$

$$\frac{\emptyset \vdash\ v\ :\ \tau_2 \qquad \tau_2 \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash\ \mathsf{F}\ ::\ \mathsf{H}\ :\ \tau \qquad \vdash\ \mathsf{K}\ :\ \tau}{\vdash\ \langle\ \mathbf{resume}(\mathsf{F}\ ::\ \mathsf{H})(v)\ \|\ \mathsf{K}\ \rangle\ ok}\ [\textsc{Rewinding}]$$

Runtime Expression Typing

$$\frac{u_0\ =\ w\ +\!\!+\ u_1}{\emptyset \vdash\ w\ :\ u_0\ \sqsubseteq\ u_1}\ [\textsc{Evidence}] \qquad \frac{\Gamma,\ x :\ \tau_1,\ k :\ \tau_2 \to u\ \tau \mid u \vdash\ s\ :\ \tau}{\emptyset \vdash\ \mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}\ :\ \mathbf{Cap}\ (l\ ::\ u)\ \tau_1\ \tau_2}\ [\textsc{Capability}]$$

$$\frac{\tau_1 \mid \rho \vdash\ \mathsf{F}\ ::\ \mathsf{H}\ :\ \tau_2}{\emptyset \vdash\ \mathbf{resume}(\mathsf{F}\ ::\ \mathsf{H})\ :\ \tau_1 \to \rho\ \tau_2}\ [\textsc{Continuation}]$$

~ End Figblock

Stack Typing

$\boxed{\vdash\ \mathsf{K}\ :\ \tau}$

$$\frac{}{\vdash\ \bullet\ :\ \tau}\ [\textsc{Exit}] \quad \frac{x :\ \tau \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash\ s\ :\ \tau_1 \qquad \vdash\ \mathsf{K}\ :\ \tau_1}{\vdash\ \mathbf{val}\ x\ =\ \square;\ s\ ::\ \mathsf{K}\ :\ \tau}\ [\textsc{Frame}]$$

$$\frac{\vdash\ \mathsf{K}\ :\ \tau}{\vdash\ \#_l\ \{\ \square\ \}\ ::\ \mathsf{K}\ :\ \tau}\ [\textsc{Handler}]$$

Resumption Typing

$\boxed{\tau \mid \rho \vdash\ \mathsf{H}\ :\ \tau}$

$$\frac{}{\tau \mid \rho \vdash\ \bullet\ :\ \tau}\ [\textsc{Exit}] \qquad \frac{x :\ \tau_1 \mid \rho \vdash\ s\ :\ \tau_0 \qquad \tau \mid \rho \vdash\ \mathsf{H}\ :\ \tau_1}{\tau \mid \rho \vdash\ \mathbf{val}\ x\ =\ \square;\ s\ ::\ \mathsf{H}\ :\ \tau_0}\ [\textsc{Frame}]$$

$$\frac{\tau \mid l\ ::\ \rho \vdash\ \mathsf{H}\ :\ \tau_0}{\tau \mid \rho \vdash\ \#_l\ \{\ \square\ \}\ ::\ \mathsf{H}\ :\ \tau_0}\ [\textsc{Handler}]$$

Figure 7 Abstract machine typing of Λ_{cap}

A Abstract Machine and Simulation

A.1 Machine Typing

Figure 7 lists the typing rules of the abstract machine.

A.2 Machine Steps

Figure 8 lists the stepping rules of the abstract machine including regions and evidence which are needed to prove invariants throughout execution.

Machine Steps:

(return)	$\langle\ \mathbf{return}\ v \parallel \mathbf{val}\ x\ =\ \square;\ s\ ::\ \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ s[x \mapsto v] \parallel \mathsf{K}\ \rangle$
(push)	$\langle\ \mathbf{val}\ x\ =\ s_0;\ s \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ s_0 \parallel \mathbf{val}\ x\ =\ \square;\ s\ ::\ \mathsf{K}\ \rangle$
(call)	$\langle\ \{\ [\overline{r}\ ;\ \overline{n : \gamma}](\overline{x : \tau})\ \mathbf{at}\ \rho \Rightarrow s_0\ \}[\overline{\rho}, \overline{e}](\overline{v}) \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ s_0[\overline{r \mapsto \rho}, \overline{n \mapsto e}, \overline{x \mapsto v}] \parallel \mathsf{K}\ \rangle$
(try)	$\langle\ \mathbf{try}\ \{\ [r\ ;\ n](c) \Rightarrow s_0\ \}\ \mathbf{with}\ \{\ (x,\ k) \Rightarrow s\ \} \parallel \mathsf{K}\ \rangle$ where $l\ =\ \texttt{generateFresh}()$, and $u\ =\ l\ ::\ \mathcal{R}[\![\ \mathsf{K}\ ]\!]$,\$ and $w\ =\ l\ ::\ \bullet$, and $v\ =\ \mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}$	$\rightarrow$	$\langle\ s_0[r \mapsto u, n \mapsto w, c \mapsto v] \parallel \#_l\ \{\ \square\ \}\ ::\ \mathsf{K}\ \rangle$
(pop)	$\langle\ \mathbf{return}\ v \parallel \#_l\ \{\ \square\ \}\ ::\ \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ \mathbf{return}\ v \parallel \mathsf{K}\ \rangle$
(perform)	$\langle\ \mathbf{do}\ (\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[e](v) \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ \mathbf{do}\ (\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[\ \mathcal{N}[\![e]\!]\ ](v) \parallel \mathsf{K} \parallel \bullet\ \rangle$
(unwind)	$\langle\ \mathbf{do}\ (\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[w](v) \parallel \mathsf{F}\ ::\ \mathsf{K} \parallel \mathsf{H}\ \rangle$ where $\mathsf{F}\ =\ \mathbf{val}\ x\ =\ \square;\ s$	$\rightarrow$	$\langle\ \mathbf{do}\ (\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[w](v) \parallel \mathsf{K} \parallel \mathsf{F}\ ::\ \mathsf{H}\ \rangle$
(forward)	$\langle\ \mathbf{do}\ (\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[l'\ ::\ w](v) \parallel \mathsf{F}\ ::\ \mathsf{K} \parallel \mathsf{H}\ \rangle$ where $\mathsf{F}\ =\ \#_{l'}\ \{\ \square\ \}$, and $l \neq l'$	$\rightarrow$	$\langle\ \mathbf{do}\ (\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[w](v) \parallel \mathsf{K} \parallel \mathsf{F}\ ::\ \mathsf{H}\ \rangle$
(handle)	$\langle\ \mathbf{do}\ (\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[\bullet](v) \parallel \mathsf{F}\ ::\ \mathsf{K} \parallel \mathsf{H}\ \rangle$ where $\mathsf{F}\ =\ \#_l\ \{\ \square\ \}$, and $j\ =\ \mathbf{resume}(\mathsf{F}\ ::\ \mathsf{H})$	$\rightarrow$	$\langle\ s[x \mapsto v][k \mapsto j] \parallel \mathsf{K}\ \rangle$
(rewind)	$\langle\ \mathbf{resume}(\mathsf{F}_1\ ::\ \mathsf{F}_2\ ::\ \mathsf{H})(v) \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ \mathbf{resume}(\mathsf{F}_2\ ::\ \mathsf{H})(v) \parallel \mathsf{F}_1\ ::\ \mathsf{K}\ \rangle$
(resume-1)	$\langle\ \mathbf{resume}(\mathbf{val}\ x\ =\ \square;\ s\ ::\ \bullet)(v) \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ s[x \mapsto v] \parallel \mathsf{K}\ \rangle$
(resume-2)	$\langle\ \mathbf{resume}(\#_l\ \{\ \square\ \}\ ::\ \bullet)(v) \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ \mathbf{return}\ v \parallel \mathsf{K}\ \rangle$

Figure 8 Steps of the abstract machine.

A.3 Translation of Machine States

Figure 9 lists the translation of machine states to System F. While the translation of the source program is straight-forward, to translate intermediate steps, we need to define additional translation functions and auxiliary contexts.

Auxiliary Contexts

To define the translation, we add auxiliary contexts C. Contexts C are either empty $\bullet$, or an application in a larger context $\square\ v$, or a binding of a special continuation variable κ in a larger context (that is, $\mathbf{let}\ \kappa\ =\ v\ \mathbf{in}\ \square$). Plugging a term into a context is straight-forward. The case of the continuation binder performs substitution during plugging. Substitutions always happen on the meta-level during translation and never occur in translated terms.

Translation of Machine States

The translation of execution machine states (that is, $\langle\ s \parallel \mathsf{K}\ \rangle$) plugs the translated statement into a context which applies it to the continuation variable κ, and then into the translation of the stack $\mathcal{K}[\![\ \mathsf{K}\ ]\!]$. This will bind κ to the actual continuation. The translation of the unwinding machine states, that is

$$\langle\ \mathbf{do}\ \mathbf{cap}_l\{(x, k) \Rightarrow s\}[w](v) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle$$

plugs the unwinding term (that is, $\mathcal{W}[\![\ w\ ]\!]_{(\lambda k.\ \mathcal{S}[\![s]\!])[x\ \mapsto\ \mathcal{V}[\![v]\!]]}$) into an application to the translation of the stack trace ($\square\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]$), and then into the translation of the stack ($\mathcal{K}[\![\ \mathsf{K}\ ]\!]$).

Syntax of System **F:**

Terms
$t \quad ::= \quad x \mid \lambda x.\ t \mid \Lambda a.\ t \mid t\ t \mid t\ \tau \mid \texttt{done}$

Contexts
$C \quad ::= \quad \bullet \mid \Box\ v\ ::\ C \mid \textbf{let}\ \kappa\ =\ v\ \textbf{in}\ \Box\ ::\ C$

Plugging
$\textsc{Plug}(t,\ \bullet) = t$
$\textsc{Plug}(t,\ \Box\ v\ ::\ C) = \textsc{Plug}(t\ v,\ C)$
$\textsc{Plug}(t,\ \textbf{let}\ \kappa\ =\ v\ \textbf{in}\ \Box\ ::\ C) = \textsc{Plug}(t\ [\kappa \mapsto v],\ C)$

Translation of Machine States:

$\mathcal{M}[\![\ \langle\ s \parallel \mathsf{K}\ \rangle\ ]\!] = \textsc{Plug}(\mathcal{S}[\![\ s\ ]\!],\ \Box\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!])$
$\mathcal{M}[\![\ \langle\ \textbf{do}\ \textbf{cap}_l\{\ (x,k) \Rightarrow s\ \}[w](v) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle\ ]\!] =$
$\quad \textsc{Plug}(\mathcal{W}[\![\ w\ ]\!]_{(\lambda k.\ \mathcal{S}[\![s]\!])[x\ \mapsto\ \mathcal{V}[\![v]\!]\ ])},\ \Box\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!])$
$\mathcal{M}[\![\ \langle\ \textbf{resume}(\mathsf{F}\ ::\ \mathsf{H})(v) \parallel \mathsf{K}\ \rangle\ ]\!] = \textsc{Plug}(\mathcal{H}[\![\mathsf{H}]\!]\ \mathcal{V}[\![v]\!],\ \mathcal{K}[\![\mathsf{F}\ ::\ \mathsf{K}]\!])$

Translation of Stacks:

$\mathcal{K}[\![\ \bullet\ ]\!] = \textbf{let}\ \kappa\ =\ \texttt{done}\ \textbf{in}\ \Box\ ::\ \bullet$
$\mathcal{K}[\![\ \textbf{val}\ x\ =\ \Box;\ s\ ::\ \mathsf{K}\ ]\!] = \textbf{let}\ \kappa\ =\ \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa\ \textbf{in}\ \Box\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]$
$\mathcal{K}[\![\ \#_l\ \{\ \Box\ \}\ ::\ \mathsf{K}\ ]\!] = \textbf{let}\ \kappa\ =\ \lambda x_1.\ \lambda k_1.\ k_1\ x_1\ \textbf{in}\ \Box\ ::\ \Box\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]$

Translation of Stack Traces:

$\mathcal{H}[\![\ \bullet\ ]\!] = \kappa$
$\mathcal{H}[\![\ \textbf{val}\ x\ =\ \Box;\ s\ ::\ \mathsf{H}\ ]\!] = \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa]$
$\mathcal{H}[\![\ \#_l\ \{\ \Box\ \}\ ::\ \mathsf{H}\ ]\!] = \lambda x_0.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \lambda x_1.\ \lambda k_1.\ k_1\ x_1]\ x_0\ \kappa$

Translation of Unwinding:

$\mathcal{W}[\![\ \bullet\ ]\!]_t = t$
$\mathcal{W}[\![\ l\ ::\ w\ ]\!]_t = \lambda k.\ \lambda j.\ \mathcal{W}[\![\ w\ ]\!]_t\ (\lambda x.\ k\ x\ j)$

Translation of Evidence Values:

$\mathcal{E}[\![\ w\ ]\!] = \Lambda a.\ \lambda m.\ \mathcal{W}[\![\ w\ ]\!]_m$

Translation of Runtime Values:

$\mathcal{V}[\![\ \textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}\ ]\!] = \lambda x.\ \lambda k.\ \mathcal{S}[\![\ s\ ]\!]$
$\mathcal{V}[\![\ \textbf{resume}(\#_l\ \{\ \Box\ \}\ ::\ \mathsf{H})\ ]\!] = \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \lambda x_1.\ \lambda k_1.\ k_1\ x_1]$
$\mathcal{V}[\![\ \textbf{val}\ x\ =\ \Box;\ s\ ::\ \mathsf{H})\ ]\!] = \mathit{dummy}(\mathit{does\ not\ occur})$

Figure 9 Translation of machine states.

The translation of the stack trace contains κ free exactly once, which will be bound by the translation of the stack. The overall term is always closed.

A.4 Proof of Simulation

To prove simulation, we need the following lemma that translation commutes with substitution:

▶ **Lemma 13.** $\mathcal{S}[\![\ s\ ]\!]\ [x \mapsto \mathcal{V}[\![\ v\ ]\!]\ ] = \mathcal{S}[\![\ s\ [x \mapsto v]\ ]\!]$

And mutadis mutandis for regions and evidence.

We also need the following lemma to prove that applying a translated evidence term produces an unwinding term.

▶ **Lemma 14** (Perform). $\mathcal{E}[\![\ e\ ]\!]\ \tau\ t \to^* \mathcal{W}[\![\ \mathcal{N}[\![\ e\ ]\!]\ ]\!]_t$

Proof. The proof proceeds by induction on evidence terms. ◀

Given the above lemmas, we can prove simulation:

Proof of Theorem 10 (Simulation). The proof proceeds by case analysis of the step taken by the abstract machine. Most of them are straight-forward, except for *(perform)*, which requires Lemma 14:

$$
\begin{array}{lcl}
\mathcal{M}[\![\ \langle\ \textbf{do cap}_l\{(x,k)\Rightarrow s\}[e](v)\ \|\ \mathsf{K}\ \rangle\ ]\!] & = & \text{by Definition } \mathcal{M}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{S}[\![\ \textbf{do cap}_l\{(x,k)\Rightarrow s\}[e](v)\ ]\!],\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = & \text{by Definition } \mathcal{S}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{E}[\![\ e\ ]\!]\ \mathcal{T}[\![\tau_2]\!]\ (\mathcal{V}[\![\textbf{cap}_l\{(x,k)\Rightarrow s\}]\!]\ \mathcal{V}[\![v]\!]),\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = & \text{by Definition } \mathcal{V}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{E}[\![\ e\ ]\!]\ \mathcal{T}[\![\tau_2]\!]\ ((\lambda x.\ \lambda k.\ \mathcal{S}[\![s]\!])\ \mathcal{V}[\![v]\!]),\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & \to & \text{by beta reduction} \\
\textsc{Plug}(\mathcal{E}[\![\ e\ ]\!]\ \mathcal{T}[\![\tau_2]\!]\ ((\lambda k.\ \mathcal{S}[\![s]\!])\ [x \mapsto \mathcal{V}[\![v]\!]\ ]),\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & \to^* & \text{by Lemma } \textsc{Perform} \\
\textsc{Plug}(\mathcal{W}[\![\ \mathcal{N}[\![\ e\ ]\!]\ ]\!]_{((\lambda k.\ \mathcal{S}[\![s]\!])\ [x \mapsto \mathcal{V}[\![v]\!]\ ])},\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = & \text{by Definition } \mathcal{H}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{W}[\![\ \mathcal{N}[\![\ e\ ]\!]\ ]\!]_{((\lambda k.\ \mathcal{S}[\![s]\!])\ [x \mapsto \mathcal{V}[\![v]\!]\ ])},\ \square\ \mathcal{H}[\![\ \bullet\ ]\!]\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = & \text{by Definition } \mathcal{M}[\![\ \cdot\ ]\!] \\
\mathcal{M}[\![\ \langle\ \textbf{do cap}_l\{(x,k)\Rightarrow s\}[\ \mathcal{N}[\![e]\!]\ ](v)\ \|\ \mathsf{K}\ \|\ \bullet\ \rangle\ ]\!] & & \\
\end{array}
$$

◀