Region-based Resource Management in Continuation-Passing Style

Philipp Schuster
University of Tübingen, Germany

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Klaus Ostermann
University of Tübingen, Germany

Abstract

Regions are a useful tool for the safe and automatic management of scarce resources. Due to their scarcity, resources are often limited in their lifetime which is associated with a certain scope. When control flow leaves the scope, the resources are destroyed. Exceptions can non-locally exit such scopes and it is important that resources are also destroyed in this case. Continuation-passing style is a useful compiler intermediate language that makes control flow explicit. All calls are tail calls and the runtime stack is not used. It can also serve as an implementation technique for control effects like exceptions. In this case throwing an exception means jumping to a continuation which is not the return continuation. How is it possible to offer region-based resource management and exceptions in the same language and translate both to continuation-passing style? In this paper, we answer this question. We present a typed language with resources and exceptions, and its translation to continuation-passing style. The translation can be defined modularly for resources and exceptions – the correct interaction between the two automatically arises from simple composition. We prove that the translation preserves well-typedness and semantics.

First Published January, 2nd 2022
URL https://se.informatik.uni-tuebingen.de/publications/schuster22region
Comments This is a technical report. The definitive version has been accepted to appear in the Proceedings of the 31st European Symposium on Programming (ESOP) 2022.

1 Introduction

Regions were originally introduced for the safe and automatic management of memory [28]. Since then, much research has happened to extend their usefulness for memory management in different scenarios [13, 8, 12, 11]. Regions are also a useful tool for controlling the allocation, release, and use of any kind of scarce resource even when considering memory to be plentiful [18]. Resources are organized into a stack of regions which corresponds to nested scopes in the program. Resources in a region are automatically released when control flow leaves the corresponding scope. A type-and-region system guarantees *resource safety*, *i.e.*, that there is no access to a resource outside of its corresponding scope.

Exceptions allow for non-local exits from scopes. It is important that resources are released not only upon normal return, but also when an exception is thrown. A type-and-effect system statically ensures that certain error conditions do not occur when running a program. In the case of exceptions, for example, we want to guarantee *exception safety*, *i.e.*, every exception is eventually caught. Some work on regions explicitly caters to exceptions [18, 13, 29, 17]. Still, the interaction between regions, exceptions, and first-class functions is non-trivial. To the best of our knowledge region safety for a language with this combination of features has not yet been formally established.

Continuation-passing style (CPS) is an attractive [2, 16, 7] intermediate representation for programs. Control flow is explicit, and many program optimizations amount to simple inlining and beta reduction. CPS can also be an implementation technique for control effects like exceptions [16, 15, 25]. Optimization of programs using these features still amounts

to inlining and reduction. In CPS all calls are tail calls. Importantly, *there is no runtime stack* that a thrown exception unwind. Instead, throwing an exception means jumping to a non-return continuation.

A CPS translation (from a source to a target language in CPS) must of course be correct, *i.e.* preserve the semantics of the source language. Ideally, the target language is also typed, and the translation takes well-typed terms to well-typed terms. Moreover, when we translate a source program with exceptions to CPS, well-typedness of the target term should also entail exception safety. While there are CPS translations for exceptions, and it is translate resource bracketing to CPS, there is not yet a single CPS translation for both features in the same language. Moreover, it is not clear how such a combination could be typed in a way that guarantees resource safety and exception safety at the same time.

We present an intermediate language Λ_ρ with resources and exceptions. It has a type-and-effect system keeping track of regions to model both: the lifetime of resources as well as the scope of exception handlers. We define its operational semantics as an instrumented [22] abstract machine, which manipulates a runtime stack. We prove progress (Theorem 4) and preservation (Theorem 5) for this semantics in the proof assistant Coq. Resource safety (Theorem 6) and exception safety (Theorem 7) follow as corollaries. To our knowledge, this is the first proof of safety for a language with region-based resource management, exceptions, and first-class functions.

We define a CPS translation from Λ_ρ to System F with base types and primitive operations. The translation takes well-typed terms to well-typed terms (Theorem 12). We implemented the translation as a shallow embedding into the dependently typed language Idris 2. It does not use any special runtime constructs, neither for regions nor for exceptions. The translation is correct: translated terms simulate the abstract machine semantics step-wise (Theorem 14). This entails resource safety and exception safety for CPS translated terms.

Our key technical idea is to understand regions as describing the runtime stack. In the operational semantics, language constructs for resources and exceptions push freshly generated markers onto the runtime stack. At runtime, a region stands for the concrete list of markers on the stack. Subregioning evidence to stand for the concrete difference between two such lists. In CPS there is no stack. Under our CPS translation, regions are answer types, and subregioning evidence terms are answer-type coercing functions. They move from one region to another one. This allows us to define the CPS translations of resource management and exceptions separately while having them interact correctly.

The rest of the paper is organized as follows. In Section 2, we introduce the main ideas behind our language Λ_ρ. In Section 4, we formally present Λ_ρ. We start with a base language with type-level region tracking and term-level subregioning evidence. We gradually extend this base language with region-based resource management and exceptions. In Section 5, we define the CPS translation for Λ_ρ to System F. We do so gradually, first for the base language, then for resources, then for exceptions. In Section 6 we compare to related work and in Section 7 we summarize the key ideas and outline future work.

2 Overview

Here, we provide an informal overview of our main ideas and the language Λ_ρ. We start by re-iterating how regions are used for resource management. We then introduce exceptions and show how we translate them to CPS. Finally, we combine resources and exceptions and demonstrate how our translation reveals information about the use of resources in the presence of non-local exits.

2.1 Regions for Resources

As a first example, let us see how regions can be used to manage file handles in Λ_ρ. Our type system follows Fluet and Morrisett [11] and Kiselyov and Shan [18] with some minor differences.

▶ **Example 1.** Consider the following simple example, which copies the first line of a file `"input"` into a file `"output"` and additionally inserts a line at the beginning and a line at the end of the output file. Both files are automatically closed and any attempt, accidental or not, to use them after they are closed will fail.

```
pool { [r1](p1 : Pool r1, l1 : r1 ⊑ Top) ⇒
  val out: File r1 = open(p1, "output", 𝟘);
  writeln(out, "start", 𝟘);
  pool { [r2](p2 : Pool r2, l2 : r2 ⊑ r1) ⇒
    val in: File r2 = open(p2, "input", 𝟘);
    val firstLine = readln(in, 𝟘);
    writeln(out, firstLine, l2)
  };
  writeln(out, "end", 𝟘);
  return ()
}
```

We use a `pool { ... }` statement to create a fresh resource pool. A pool is a reference to a list of open files. All files in this list are automatically closed when control flow leaves the enclosed block. The `pool` statement introduces a *region variable* `r1`, a *pool variable* `p1` and *subregioning evidence* `l1`. We then open the file `"output"` in pool `p1`. In our type system, every statement is checked in a region. The overall statement is checked in the top-level region `Top`. The enclosed block is checked in region `r1`. When we open a file, we have to explicitly pass evidence that the region of the pool is a subregion of the current region. In this example, we pass the reflexivity evidence `𝟘 : r1 ⊑ r1`. We create a second pool `p2` in a second region `r2`, which is clearly inside of `r1`. This fact is witnessed by the evidence variable `l2`. When we write to the output file, we have to provide evidence that the file's region `r1` is inside of the current region `r2`. We provide `l2 : r1 ⊑ r2`.

For this simple example, after applying our CPS translation and some beta reduction we get the following straight-line code.

```
λk.
  let p1 = createPool ();
  let out = openFile p1 "output";
  writeLine out "start";
  let p2 = createPool ();
  let in = openFile p2 "input";
  let firstLine = readLine in;
  writeLine out firstLine;
  destroyPool p2;
  writeLine out "end";
  destroyPool p1;
  k ()
```

The original progam did not contain any interesting control flow and our CPS translation results in a sequence of primitive operations. There is no overhead for protecting resources when no exception is thrown. Later we will see how we clean up resources when there are exceptions. But first, let us look at our CPS translation of exceptions.

2.2 Regions for Exception Handlers

Exceptions abort the current computation to an exception handler. An exception that is thrown while the corresponding handler is not on the stack results in an error condition that we statically prevent from happening. In Λ_ρ, we use the same mechanism for resources and exceptions and enforce *exception safety* in terms of regions: in order to throw to an exception handler, we require evidence that the corresponding handler is still on the stack.

Exceptions in Λ_ρ are lexically scoped [32, 31, 4, 5]. This style of exceptions has advantages when reasoning about higher-order functions. Operationally, each `try` statement generates a fresh marker at runtime and pushes a catch frame with this marker onto the stack. We explicitly pass these markers as values of type `Catch r`. For example, consider the following program.

▶ **Example 2.** The function `safeDiv` divides two numbers, but throws an exception when the second number is zero.

```
def safeDiv[r](x : Int, y : Int, e : Catch r) at r {
  if (y == 0) { throw(e, 𝟘) }
  else { return (x / y) }
}
```

In addition to the two parameters `x` and `y`, the function `safeDiv` receives a catch marker `e`. When `y` is zero we throw to `e`. For this to be safe we need to guarantee that we only throw to `e` in the dynamic extent of the corresponding exception handler. But this is the very same problem we had with pools. So we use the very same solution: When we throw to a catch frame of type `Catch r` we have to provide evidence that the current region is a subregion of the catch's region, in this example $\mathbb{0}$ `: r` $\sqsubseteq$ `r`.

The function `safeDiv` is *region polymorphic*. It abstracts over a region variable `r`. It is also annotated to run in the region `r`. To handle the exception we use our `safeDiv` function as follows.

```
try { [r1](e1 : Catch r1, l1 : r1 ⊑ Top) ⇒
  safeDiv[r1](5, 0, e1)
} catch { return 0 }
```

Very much like the `pool` statement, the exception handler introduces a region variable `r1`, a handler `e1`, and subregioning evidence `l1`. In the call to `safeDiv`, we instantiate the region variable `r` to `r1` and pass the exception handler `e1`. The example illustrates that we can guarantee exception safety by the very same mechanism we use for region safety.

When we translate this program to CPS, inline the function `safeDiv`, and after applying beta reduction and commuting conversions we get the following:

$$\lambda k_2.\ \textbf{if}\ (0 \equiv 0)\ \textbf{then}\ k_2\ 0\ \textbf{else}\ k_2\ (5\ /\ 0)$$

When we translate programs to CPS, control flow becomes explicit. This is also true in the presence of control effects like exceptions. Because of this, optimizing programs in CPS amounts to beta reduction. How then can we achieve the same in the presence of resources *and* exceptions?

2.3 Combining Resources and Exceptions

Consider the following simple program that mixes pools and exceptions.

▶ **Example 3.** We install an exception handler and create two resource pools. We open a file in the inner pool, open a file in the outer pool, and then throw an exception.

```
try { [r1](e1 : Catch r1, l1 : r1 ⊑ Top) ⇒
  pool { [r2](p2 : Pool r2, l2 : r2 ⊑ r1) ⇒
    pool { [r3](p3 : Pool r3, l3 : r3 ⊑ r2) ⇒
      open(p3, "input", 𝟘);
      open(p2, "output", l3);
      throw(e1, l3 ⊕ l2)
    }
  }
} catch { return 1 }
```

To open files into pools, we have to provide evidence, as before. To throw an exception to the outer handler `e1`, we have to provide evidence that region `r3` is inside of `r1`. We *compose* evidence variables `l3` ⊕ `l2`, to get evidence of type `r3` ⊑ `r1`.

This program, after CPS translation, reduces to the following program. The exception handler is known and will be eliminated. Again, simplifying control flow amounts to beta reduction as usual in CPS.

```
λk.
  let p2 = createPool ();
  let p3 = createPool ();
  openFile p3 "input";
  openFile p2 "output";
  destroyPool p3;
  destroyPool p2;
  k 1
```

In our framework, these simplifications of control flow also correctly account for proper creation and destruction of resources. We can blindly reduce the translated program without any extra considerations.

3 First-Class Functions

Λ_ρ fully supports first-class functions. For example, consider the following program which factors out a common pattern as a higher-order function.

```
def withFile[r0](path: String, f: [r](File r, r ⊑ r0) →r Unit) at r0 {
  pool { [r1](p1: Pool r1, l1: r1 ⊑ r0) ⇒
    val file = open(p1, path, 𝟘);
    f[r1](file, l1)
  }
}
```

The function `withFile` is region polymorphic. It abstracts over the region `r0` it can be used in. The function `f` must be region-polymorphic too, because we use it under a new region `r1`. We instantiate its region parameter with `r1` pass evidence `l1`.

It would be possible to write `withFile` with the following signature:

```
withFile : [r0](path: String, f: [r](File r) →r Unit) →r0 Unit
```

Here, the function parameter `f` would not receive any evidence. This variant of `withFile` would be less useful, as `f` could not access any resources from outside of the call-site of `withFile`.

Terms:

Statements

s	::=	**val** $x = s; s$	sequencing
	|	**return** e	returning
	|	$e[\overline{\rho}](\overline{e})$	application

Expressions

$e,\ i$	::=	$x,\ f,\ l,\ ...$	variables
	|	v	values
	|	$\mathbb{0}$	reflexivity ev.
	|	$e \oplus e$	transitivity ev.

Values

v	::=	() | 0 | 1 | ... | true | ...	primitives
	|	$\{\ [\overline{r}](\overline{x\ :\ \tau})$ **at** $\rho \Rightarrow s\}$	closures

Types:

Types

τ	::=	Int | Bool | ...	primitives
	|	$\forall[\overline{r}]\ (\overline{\tau}) \to^{\rho} \tau$	functions
	|	$\rho \sqsubseteq \rho$	evidence

Regions

ρ	::=	r	region variable
	|	$\top$	toplevel region

Environments:

Γ	::=	$\emptyset$	empty env.
	|	$\Gamma,\ r$	region binding
	|	$\Gamma,\ x\ :\ \tau$	value binding

Names:

$x,\ y,\ l$	::=	x | y | l ...	value variables	r	::=	r | s | ...	region variables

Figure 1 Syntax of the core of Λ_ρ.

4 A Language with Regions, Resources, and Exceptions

In this section, we formally present Λ_ρ and its operational semantics. We will introduce Λ_ρ step-by-step starting with a base language with support for type-level region tracking but no interesting term-level features that make use of them. We then add resource pools, exceptions, and finally consider the combination of the two. The operational semantics is given in terms of an abstract machine that manipulates are runtime stack. In Section 5, we present a CPS translation of Λ_ρ, following the same incremental development.

The paper is accompanied by a mechanized formalization of Λ_ρ and its operational semantics in the Coq theorem prover [3], including the usual theorems of Progress (Theorem 4) and Preservation (Theorem 5). Resource- and exception safety follow as corollaries: whenever we use a resource (like a file) it is live (Corollary 6), and whenever we throw an exception the corresponding handler is on the stack (Corollary 7).

Our operational semantics will push freshly generated markers onto the runtime stack. A region is the list of concrete *markers on the stack* and evidence is the list of markers that is the *difference* between two such lists. Although they do not play any role computationally, for our proofs we will substitute these lists for region variables and evidence variables at runtime. Our typing rules for runtime evidence then makes proving region safety and exception safety possible.

4.1 Syntax

Figure 1 defines the syntax of the core of Λ_ρ. We use fine-grain call-by-value [21] and syntactically distinguish between statements, which can have effects, and pure expressions.

Function values (*i.e.*, $\{\ [\overline{r}](\overline{x\ :\ \tau})$ **at** $\rho \Rightarrow s\}$) abstract over a list of type-level region parameters (*i.e.*, $\overline{r}$), and a list of term-level parameters (*i.e.*, $\overline{x\ :\ \tau}$). Each function is defined to run exactly in a region ρ, but otherwise functions are unsurprising. Since our focus is on the interaction between regions and control effects, we omit type abstraction from this

Statement Typing

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

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

$$\frac{\Gamma \vdash\ e_0\ :\ \forall[\overline{r}](\overline{\tau}) \to^{\rho_0} \tau_0 \qquad \overline{\Gamma \vdash\ e\ :\ \tau[\overline{r \mapsto \rho}]} \qquad \rho\ =\ \rho_0[\overline{r \mapsto \rho}]}{\Gamma \mid \rho \vdash\ e_0[\overline{\rho}](\overline{e})\ :\ \tau_0[\overline{r \mapsto \rho}]}\ [\textsc{App}]$$

Expression Typing

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

$$\frac{\Gamma(x)\ =\ \tau}{\Gamma \vdash\ x\ :\ \tau}\ [\textsc{Var}] \quad \frac{}{\Gamma \vdash\ n\ :\ \mathsf{Int}}\ [\textsc{Lit}] \quad \frac{\Gamma,\ \overline{r},\ \overline{x : \tau} \mid \rho \vdash\ s_0\ :\ \tau_0}{\Gamma \vdash\ \{\ [\overline{r}](\overline{x\ :\ \tau})\ \textsf{at}\ \rho \Rightarrow s_0\ \}\ :\ \forall[\overline{r}](\overline{\tau}) \to^{\rho} \tau_0}\ [\textsc{Fun}]$$

$$\frac{}{\Gamma \vdash\ \mathbb{0}\ :\ \rho\ \sqsubseteq\ \rho}\ [\textsc{Reflexive}] \qquad \frac{\Gamma \vdash\ e\ :\ \rho\ \sqsubseteq\ \rho' \qquad \Gamma \vdash\ e'\ :\ \rho'\ \sqsubseteq\ \rho''}{\Gamma \vdash\ e\ \oplus\ e'\ :\ \rho\ \sqsubseteq\ \rho''}\ [\textsc{Transitive}]$$

Figure 2 Type system of the core of Λ_ρ.

presentation. Our mechanization includes type polymorphism, which is orthogonal to the rest of the calculus. We define the following short-hand notation for named function definitions:

$$\textsf{def}\ f[\overline{r}](\overline{x\ :\ \tau})\ \textsf{at}\ \rho\ \{\ s_0\ \};\ s \qquad \doteq \qquad \textsf{val}\ f\ =\ \textsf{return}\ \{\ [\overline{r}](\overline{x\ :\ \tau})\ \textsf{at}\ \rho \Rightarrow s_0\};\ s$$

The list of region parameters scopes over the parameter types, the return type, the annotated region ρ, and the body of function s. We apply functions to a list of regions $\overline{\rho}$ and a list of arguments $\overline{e}$.

We introduce two additional concepts: type-level regions and term-level evidence. Type-level regions ρ are either region variables r or the top-level region $\top$. Intuitively, the top-level region denotes the bottom part of the runtime stack. Term-level evidence expressions are either the empty evidence $\mathbb{0}$ witnessing reflexivity of subregioning, or the composition of evidence $e \oplus e$, witnessing the transitivity of subregioning. By convention, we use the meta-variables f and l to stand for variables of function type and evidence type respectively, and we use the meta-variable i to stand for expressions of evidence type.

4.2 Typing

Figure 2 defines typing of core Λ_ρ. We type statements and expressions with different judgement forms. While both are typed in an environment Γ containing value and region 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 specific regions. In contrast, expressions are pure (that is, *trivial*) and can be evaluated independent of any region.

Syntax of the Abstract Machine:

Machine States	Stacks	Frames
$\mathsf{M} ::= \langle\, s \parallel \mathsf{K} \,\rangle$	$\mathsf{K} ::= \bullet \mid \mathsf{F} :: \mathsf{K}$	$\mathsf{F} ::= \mathbf{val}\ x = \square;\ s$

Machine Steps:

$$\begin{array}{lll}
\textit{(return)} & \langle\, \mathbf{return}\ e \parallel \mathbf{val}\ x = \square;\ s :: \mathsf{K} \,\rangle & \to \langle\, s[x \mapsto e] \parallel \mathsf{K} \,\rangle \\
\textit{(push)} & \langle\, \mathbf{val}\ x = s_0;\ s \parallel \mathsf{K} \,\rangle & \to \langle\, s_0 \parallel \mathbf{val}\ x = \square;\ s :: \mathsf{K} \,\rangle \\
\textit{(call)} & \langle\, \{\ [\overline{r}](\overline{x : \tau})\ \mathbf{at}\ \rho \Rightarrow s_0\ \}[\overline{\rho}](\overline{e}) \parallel \mathsf{K} \,\rangle & \to \langle\, s_0[\overline{r \mapsto \rho}][\overline{x \mapsto e}] \parallel \mathsf{K} \,\rangle
\end{array}$$

Extended Syntax:

v	$::=$ ... $\mid w$		evidence value
ρ	$::=$ ... $\mid u$		runtime region

Runtime Regions and Evidence.:

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

Runtime Region of Stack:

$$\begin{array}{lcl}
\mathcal{R}[\![\ \cdot\ ]\!] & : & \mathsf{K} \to u \\
\mathcal{R}[\![\ \bullet\ ]\!] & = & \bullet \\
\mathcal{R}[\![\ \mathbf{val}\ x = \square;\ s :: \mathsf{K}\ ]\!] & = & \mathcal{R}[\![\ \mathsf{K}\ ]\!]
\end{array}$$

Evaluation of Evidence:

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

Figure 3 Abstract machine semantics of core Λ_ρ.

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 of a computation (rule RET) can be typed in any region. In rule APP we apply a function e_0 to a list of regions $\overline{\rho}$ and to a list of arguments $\overline{e}$. The type of e_0 is a function type in 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 regions $\overline{\rho}$ for the region variables $\overline{r}$ both have to syntactically be the same. Note that we do not have any implicit or explicit subtyping of function types here or elsewhere. All region subtyping exclusively occurs through the passing of subregioning evidence.

Typing of Expressions The typing rules for variables VAR and primitives LIT are standard. Rule FUN types functions. We type the body of the function s_0 in an environment extended with the region parameters $\overline{r}$ and value parameter types $\overline{x : \tau}$. Every function is annotated with a region ρ that specifies *exactly* the region it will have to be called in. This region ρ is also the region we type the body s_0 in. 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. Value parameters of evidence type allow us to write region-polymorphic functions that are *constrained* to run in a subregion that meets these constraints.

Reflexivity evidence $\mathbb{0}$ witnesses that every region is nested within itself, and evidence $e \oplus e'$ witnesses the transitivity of nesting, which is reflected in their typing rules. We require the composition of evidence to be associative.

4.3 Operational Semantics

Figure 3 presents the operational semantics of core Λ_ρ. A machine state $\langle\, s \parallel \mathsf{K} \,\rangle$ consists of the statement s under evaluation and the runtime stack K. For the core of Λ_ρ, the stack

Syntax:

Statements

$$
\begin{array}{lcll}
s & ::= & \ldots & \\
 & \mid & \textbf{pool}\ \{\ [r](x,\ l) \Rightarrow s\ \} & \text{new resource pool} \\
 & \mid & \textbf{open}(e,\ e_0,\ i) & \text{open file} \\
 & \mid & \textbf{readln}(e,\ i) & \text{read contents}
\end{array}
$$

Types

$$
\begin{array}{lcl}
\tau & ::= & \ldots \\
 & \mid & \mathsf{Pool}\ \rho \\
 & \mid & \mathsf{File}\ \rho
\end{array}
$$

Typing Rules:

$$\frac{\Gamma,\ r,\ x\ :\ \mathsf{Pool}\ r,\ l\ :\ r \sqsubseteq \rho \mid r \vdash s\ :\ \tau}{\Gamma \mid \rho \vdash\ \textbf{pool}\ \{\ [r](x,\ l) \Rightarrow s\ \}\ :\ \tau}\ [\textsc{Pool}]$$

$$\frac{\Gamma \vdash\ e\ :\ \mathsf{File}\ \rho' \quad \Gamma \vdash\ i\ :\ \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash\ \textbf{readln}(e,\ i)\ :\ \mathsf{String}}\ [\textsc{Read}]$$

$$\frac{\Gamma \vdash\ e\ :\ \mathsf{Pool}\ \rho' \quad \Gamma \vdash\ e_0\ :\ \mathsf{String} \quad \Gamma\ \vdash\ i\ :\ \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash\ \textbf{open}(e,\ e_0,\ i)\ :\ \mathsf{File}\ \rho'}\ [\textsc{Open}]$$

Figure 4 Syntax and typing rules of resource pools.

K is a list of frames of the form $\textbf{val}\ x\ =\ \square;\ s$. The reduction rules are mostly standard. The first rule *(return)* returns to the next frame on the stack. The second rule *(push)* focuses on s_0 and pushes a frame on the stack. Finally, rule *(call)* performs reduction by simultaneously substituting region arguments $\overline{\rho}$ for region variables $\overline{r}$ and trivial expressions $\overline{e}$ for term parameters $\overline{x}$. Region parameters, the annotated region ρ, and evidence terms are operationally irrelevant. As already mentioned, we need them to maintain invariants in our proofs.

The core of Λ_ρ, as presented, does not yet contain features with interesting operational behavior. While we can abstract over regions, eventually all region variables will be instantiated with the top-level region and evidence will always be the trivial evidence.

Figure 3 also defines runtime regions and evidence values in core Λ_ρ. We extend the syntax of values with evidence values w, and the syntax of regions with runtime regions u. Both are empty lists $\bullet$ for now. In the next two sections, we will extend their syntax to be lists for markers h. The toplevel region $\top$ is the empty list runtime region $\bullet$.

To connect type-level regions ρ with the concrete runtime stack K, we define a semantic function $\mathcal{R}[\![\ \cdot\ ]\!]$, which computes the runtime region of the current stack. In core Λ_ρ, the only possible runtime region is the empty list. To give meaning to evidence expressions, we define a semantic function $\mathcal{V}[\![\ \cdot\ ]\!]$. Currently the only possible evidence value is the empty list.

4.4 Resource Pools

In this subsection, we add statements for region-based resource management to Λ_ρ. As in the introduction, we use files as an example for resources. Figure 4 introduces three additional statement forms, which introduce and eliminate non-trivial evidence to assert that all files are correctly closed. The **pool** statement delimits a new region in which we run the enclosed statement s. It introduces three variables, a fresh region variable r, a variable $x\ :\ \mathsf{Pool}\ r$, and evidence $l\ :\ r \sqsubseteq \rho$, witnessing that the fresh region r is a subregion of the outer region ρ. The **open** statement receives an pool argument e, a filename e_0, and an evidence argument $i\ :\ \rho \sqsubseteq \rho'$ that witnesses that the current region ρ is nested within the pool's region ρ'. Rule Read for **readln** statements is similar.

Syntax of Frames:

$$\mathsf{F} \quad ::= \quad \ldots \mid \#\mathbf{pool}_h\ \{\ \Box\ \} \qquad \text{resource pool frame}$$

Machine Steps:

(destroy)

$$\langle\ \mathbf{return}\ e \parallel \#\mathbf{pool}_h\ \{\ \Box\ \}\ ::\ \mathsf{K}\ \rangle \rightarrowtail \langle\ \mathbf{return}\ e \parallel \mathsf{K}\ \rangle \quad \text{do } \texttt{destroyPool}(h)$$

(pool)

$$\langle\ \mathbf{pool}\ \{\ [r](x,\ l) \Rightarrow s_0\ \} \parallel \mathsf{K}\ \rangle \quad \rightarrow\ \langle\ s_0[r \mapsto u][x \mapsto h][l \mapsto w] \parallel \#\mathbf{pool}_h\ \{\ \Box\ \}\ ::\ \mathsf{K}\ \rangle$$
$$\text{do } h\ =\ \texttt{createPool}() \qquad \text{where } u\ =\ \mathbf{po}\ h\ ::\ \mathcal{R}[\![\ \mathsf{K}\ ]\!], \text{ and } w\ =\ \mathbf{po}\ h\ ::\ \bullet$$

(open)

$$\langle\ \mathbf{open}(h,\ e,\ i) \parallel \mathsf{K}\ \rangle \quad \rightarrow\ \langle\ \mathbf{return}\ x \parallel \mathsf{K}\ \rangle \quad \text{when } \mathbf{po}\ h\ \mathit{in}\ \mathcal{R}[\![\ \mathsf{K}\ ]\!]$$
$$\text{do } x\ =\ \texttt{openFile}(h,\ e)$$

(read)

$$\langle\ \mathbf{readln}(p,\ i) \parallel \mathsf{K}\ \rangle \quad \rightarrow\ \langle\ \mathbf{return}\ x \parallel \mathsf{K}\ \rangle \quad \text{when } \mathbf{po}\ h\ \mathit{in}\ \mathcal{R}[\![\ \mathsf{K}\ ]\!]$$
$$\text{do } x\ =\ \texttt{readLine}(p) \qquad \text{where } h\ =\ p.\mathit{getPool}$$

Runtime Regions and Evidence:

$$\begin{array}{lcll} h & ::= & \texttt{@a5f} \mid \texttt{@4b2} \mid \ldots & \text{markers} \\ w & ::= & \ldots \mid \mathbf{po}\ h\ ::\ w & \text{evidence value} \\ u & ::= & \ldots \mid \mathbf{po}\ h\ ::\ u & \text{runtime region} \end{array}$$

Runtime Region of Stack:

$$\mathcal{R}[\![\ \#\mathbf{pool}_h\ \{\ \Box\ \}\ ::\ \mathsf{K}\ ]\!] \ =\ \mathbf{po}\ h\ ::\ \mathcal{R}[\![\ \mathsf{K}\ ]\!]$$

Figure 5 Abstract machine semantics of arena-based memory management.

Figure 5 extends the operational semantics. Frames can now be pool frames which contain a marker h. In rule *(pool)*, we allocate a fresh marker h and push an pool frame onto the stack. In rule *(destroy)*, we pop the pool frame and destroy the pool h, closing all associated resources. Our goal is to ensure that all access to marker h happens between these two steps.

To this end, rules *(open)* and *(read)* dynamically assert that the marker h is on the current stack K. Access to a pool that fails this test results in a stuck term. As it turns out, the mere existence of evidence i suffices to show that the assertion always succeeds (Corollary 6).

For our proof of this fact, Figure 5 extends the syntax of runtime regions and evidence. Runtime regions are now lists of pool markers and so are evidence values. The runtime region of a stack K is the list of markers that have been pushed on it. We extend the function $\mathcal{R}[\![\ \cdot\ ]\!]$ to extract this list. During execution, region variables r stand for runtime regions u. In rule *(pool)* we substitute the runtime region $\mathbf{po}\ h\ ::\ \mathcal{R}[\![\ \mathsf{K}\ ]\!]$ for the region variable r and the singleton list $\mathbf{po}\ h\ ::\ \bullet$ for the evidence variable l. Later we will see how the typing rule for evidence values connects type-level runtime regions with the concrete runtime region of the current stack K.

4.5 Exceptions

Figure 6 extends Λ_ρ with two new statement forms. The **try** ... **catch** ... statement delimits a new region in which we run the enclosed statement s_0. It introduces three variables, a fresh region variable r, a variable $x\ :\ \mathsf{Catch}\ r$, and an evidence variable $l\ :\ r\ \sqsubseteq\ \rho$, witnessing that the fresh region r is a subregion of the outer region ρ. The **throw** statement receives a

Syntax:

Statements

$$
\begin{array}{lcll}
s & ::= & \ldots & \\
 & | & \textbf{try}\ \{\ [r](x,\ l) \Rightarrow s_0\ \}\ \textbf{catch}\ \{\ s\ \} & \text{handling} \\
 & | & \textbf{throw}(e,\ i) & \text{throwing}
\end{array}
$$

Types

$$
\begin{array}{lcl}
\tau & ::= & \ldots \\
 & | & \mathsf{Catch}\ \rho\ \tau
\end{array}
$$

Typing Rules:

$$
\frac{\Gamma,\ r,\ x\ :\ \mathsf{Catch}\ r,\ l\ :\ r \sqsubseteq \rho \mid r \vdash s_0\ :\ \tau \qquad \Gamma \mid \rho \vdash s\ :\ \tau}{\Gamma \mid \rho \vdash \textbf{try}\ \{\ [r](x,\ l) \Rightarrow s_0\ \}\ \textbf{catch}\ \{\ s\ \}\ :\ \tau}\ [\textsc{Try}]
$$

$$
\frac{\Gamma \vdash e\ :\ \mathsf{Catch}\ \rho' \qquad \Gamma \vdash i\ :\ \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash \textbf{throw}(e,\ i)\ :\ \tau}\ [\textsc{Throw}]
$$

Figure 6 Syntax and typing rules of exceptions.

Syntax of Frames:

$$
\mathsf{F} \quad ::= \quad \ldots \ \mid\ \#\textbf{catch}_h\ \{\ \square\ \}\ \{\ s\ \} \qquad\qquad \text{catch frame}
$$

Machine Steps:

(popcatch)

$$
\langle\ \textbf{return}\ e \parallel \#\textbf{catch}_h\ \{\ \square\ \}\ \{\ s\ \}\ ::\ \mathsf{K}\ \rangle \ \rightarrow\ \langle\ \textbf{return}\ e \parallel \mathsf{K}\ \rangle
$$

(try)

$$
\langle\ \textbf{try}\ \{\ [r](x,\ l) \Rightarrow s_0\ \}\ \textbf{catch}\ \{\ s\ \} \parallel \mathsf{K}\ \rangle \ \rightarrow\ \langle\ s_0[r \mapsto u][x \mapsto h][l \mapsto w] \parallel \#\textbf{catch}_h\ \{\ \square\ \}\ \{\ s\ \}\ ::\ \mathsf{K}\ \rangle \quad \text{do } h\ =\ \texttt{generateFresh}()\ \text{where } u\ =\ \textbf{ca}\ h\ ::\ \mathcal{R}[\![\ \mathsf{K}\ ]\!]\ \text{and } w\ =\ \textbf{ca}\ h\ ::\ \bullet
$$

(throw)

$$
\langle\ \textbf{throw}(h,\ i) \parallel \mathsf{K}\ \rangle \ \rightarrow\ \langle\ \textbf{throw}(h,\ \mathcal{V}[\![i]\!])\ \parallel \mathsf{K}\ \rangle
$$

(unwind)

$$
\langle\ \textbf{throw}(h,\ w) \parallel \textbf{val}\ x\ =\ \square;\ s\ ::\ \mathsf{K}\ \rangle \ \rightarrow\ \langle\ \textbf{throw}(h,\ w) \parallel \mathsf{K}\ \rangle
$$

(forward)

$$
\langle\ \textbf{throw}(h,\ \textbf{ca}\ h'\ ::\ w) \parallel \#\textbf{catch}_{h'}\ \{\ \square\ \}\ \{\ s\ \}\ ::\ \mathsf{K}\ \rangle \rightarrow \langle\ \textbf{throw}(h,\ w) \parallel \mathsf{K}\ \rangle \quad \text{where } h \neq h'
$$

(catch)

$$
\langle\ \textbf{throw}(h,\ \bullet) \parallel \#\textbf{catch}_h\ \{\ \square\ \}\ \{\ s\ \}\ ::\ \mathsf{K}\ \rangle \rightarrow\ \langle\ s \parallel \mathsf{K}\ \rangle
$$

Runtime Regions and Evidence:

$$
\begin{array}{lcll}
w & ::= & \ldots \mid \textbf{ca}\ h\ ::\ w & \text{evidence value} \\
u & ::= & \ldots \mid \textbf{ca}\ h\ ::\ u & \text{runtime region}
\end{array}
$$

Runtime Region of Stack:

$$
\mathcal{R}[\![\ \#\textbf{catch}_h\ \{\ \square\ \}\ \{\ s\ \}\ ::\ \mathsf{K}\ ]\!] = \textbf{ca}\ h\ ::\ \mathcal{R}[\![\ \mathsf{K}\ ]\!]
$$

Figure 7 Abstract machine semantics of exceptions.

handler e to throw to, and evidence i that the handler's region ρ' is nested in the current region ρ.

Figure 7 extends the operational semantics. Frames can now be catch frames with a marker h and a catch statement s. In rule *(try)* we generate a fresh marker h and push a catch frame with this marker and the catch statement onto the stack. The handler x is this

Extended Machine Steps:

$$\textit{(free)} \quad \langle\, \mathsf{throw}(h,\ \mathsf{po}\ h'\ ::\ w) \parallel \#\mathsf{pool}_{h'}\ \{\ \square\ \}\ ::\ \mathsf{K}\,\rangle \to \langle\, \mathsf{throw}(h,\ w) \parallel \mathsf{K}\,\rangle \quad \text{do } \texttt{destroyPool}(h')$$

Figure 8 Abstract machine semantics of combining arenas and exceptions.

marker h. In rule *(popcatch)* we pop this catch frame upon normal return. In rule *(throw)* we transition from normal execution to unwinding. h is a handler, and $\mathcal{V}[\![\ i\ ]\!]$ evaluates the evidence expression i to a list of catch markers. In rules *(unwind)* and *(forward)* we unwind the stack frame-by-frame until we find the matching catch frame *(catch)*. Because each **try** statement generates a fresh marker at runtime, and we search for this marker during unwinding, exceptions have generative semantics [32, 31, 5, 4].

Figure 7 extends the syntax of runtime regions and evidence. They now are lists of catch markers. Again, evidence guarantees that unwinding never fails, *i.e.* the corresponding marker is always somewhere on the stack. Remarkably, we pop elements off the evidence value w in lock-step with popping catch frames off the stack and never get stuck in doing so. We always find the matching catch frame exactly when the evidence value is the empty list. The evidence value precisely reflects the list of markers between the region of the **throw** statement and the region of the **catch** statement. Importantly, this also holds for the combined language Λ_ρ (Corollary 9).

4.6 Combining Resource Pools and Exceptions

When we extend the core language with both pools and exceptions, we notice that the machine gets stuck when we would have to unwind through a pool frame. Figure 8 extends the reduction relation with this missing case. When we unwind through a $\#\mathsf{pool}_{h'}$ frame, we destroy the pool h'. In full Λ_ρ regions are lists where the elements are either an arena marker or an exception marker. Evidence is, again, the same. Having to add the rule in Figure 8 shows that under our operational semantics, the two extensions are not orthogonal. We have to explicitly consider their interaction. In Section 5, we define a CPS translation for Λ_ρ. Remarkably, both extensions can defined separately and the correct interaction automatically arises from their composition. Perhaps more importantly, the resulting terms in CPS can be reduced freely without having to consider the interaction between pools and exceptions.

4.7 Metatheory of Λ_ρ

We started out with core Λ_ρ only supporting regions and subregioning evidence. We then added two extensions, pools and exceptions, first individually, then together to arrive at the full language. Although we use resource pools for files as an example, our approach generalizes to region-based management of any resource. Indeed, in our mechanization, we do not model files and the **pool** statement only pushes and pops the fresh marker. Instead of **open** and **readln** we have a statement **check** with the following typing rule:

$$\frac{\Gamma \vdash\ e\ :\ \mathsf{Pool}\ \rho' \qquad \Gamma\ \vdash\ i\ :\ \rho\ \sqsubseteq\ \rho' \qquad \Gamma \shortmid \rho \vdash\ s\ :\ \tau}{\Gamma \shortmid \rho \vdash\ \mathsf{check}(e,\ i);\ s\ :\ \tau}\ [\textsc{Check}]$$

It asserts that the given pool is on the current runtime stack, *i.e.* live, and crashes the program if it is not. Otherwise it continues to execute statement s. We can safely access resources by first performing a runtime check and then using unsafe primitive operations. For example we would define

Abstract Machine Typing:

$$\frac{\emptyset \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash\ s\ :\ \tau \qquad \vdash\ \mathsf{K}\ :\ \tau}{\vdash\ \langle\ s\ \|\ \mathsf{K}\ \rangle\ \mathbf{ok}}\ [\textsc{Machine}]$$

Evidence Value Typing:

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

Figure 9 Abstract machine typing of Λ_ρ

$$\mathbf{open}(e,\ e_0,\ i)\ :=\ \mathbf{check}(e,\ i);\ \texttt{openFile}(e,\ e_0)$$

As we will see shortly, this check never fails.

Soundness We mechanized the formalization of Λ_ρ in the Coq theorem prover and showed the usual theorems of progress and preservation of the stepping relation on machine states M.

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

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

Figure 9 presents the typing rules for the abstract machine. An abstract machine state is well-typed when the statement s is well-typed in the concrete runtime region of the stack K. An evidence value is well-typed when it is the difference between the two runtime regions u_0 and u_1.

Properties The following properties follow directly from progress and preservation. Firstly, whenever we use a pool, it is live. The operational semantics inspects the runtime stack. But since the check always succeeds we do not have to actually perform it.

▶ **Corollary 6** (Resource Safety).
If $\langle\ \mathbf{open}(h,\ e_0,\ i)\ \|\ K\ \rangle\ \mathbf{ok}$*, then* $\mathbf{po}\ h$ *in* $\mathcal{R}[\![\ K\ ]\!]$*.*

Secondly, whenever we throw an exception, the corresponding handler is on the stack. Moreover, as we have seen from the operational semantics, during the search for the correct handler, we encounter precisely the markers that are in the evidence value.

▶ **Corollary 7** (Effect Safety).
If $\langle\ \mathbf{throw}(h,\ i)\ \|\ K\ \rangle\ \mathbf{ok}$*, then* $\mathbf{ca}\ h$ *in* $\mathcal{R}[\![\ K\ ]\!]$*.*

Thirdly, every function runs in exactly the runtime region its type requires. In other words, the type-level region ρ will at runtime stand for the concrete runtime region of the stack this function is called in.

▶ **Corollary 8** (Region Correspondence).
If $\langle\ \{\ [\overline{r}](\overline{x\ :\ \tau})\ \mathbf{at}\ \rho \Rightarrow s_0\ \}[\overline{u}](\overline{e})\ \|\ K\ \rangle\ \mathbf{ok}$*, then* $\rho[\overline{r \mapsto u}]\ =\ \mathcal{R}[\![\ K\ ]\!]$*.*

Finally, evidence values are exactly the difference between the two regions. This corollary is inspired by the similarly named theorem of Xie et al. [30].

▶ **Corollary 9** (Evidence Correspondence).
If an evidence value w *has type* $\rho_0\ \sqsubseteq\ \rho_1$*, then* ρ_0 *and* ρ_1 *are runtime regions* u_0 *and* u_1 *and* $u_0\ =\ w\ +\!\!+\ u_1$*.*

Translation of Types:

$$\begin{aligned}
&\mathcal{T}[\![\ \mathsf{Int}\ ]\!] &&= \mathsf{Int}\\
&\mathcal{T}[\![\ r\ ]\!] &&= r\\
&\mathcal{T}[\![\ \top]\!] &&= \mathsf{Void}\\
&\mathcal{T}[\![\ \forall[\overline{r}](\overline{\tau}) \to^{\rho} \tau_0\ ]\!] &&=\\
&\quad \overline{\forall r.}\ \overline{\mathcal{T}[\![\tau]\!]} \to \textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \mathcal{T}[\![\ \tau_0\ ]\!]\\
&\mathcal{T}[\![\ \rho \sqsubseteq \rho'\ ]\!] &&=\\
&\quad \forall a.\ \textsc{Cps}\ \mathcal{T}[\![\ \rho'\ ]\!]\ a \to \textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ a
\end{aligned}$$

Translation of Expressions:

$$\begin{aligned}
&\mathcal{E}[\![\ x\ ]\!] &&= x\\
&\mathcal{E}[\![\ \{\ [\overline{r}](\overline{x : \tau})\ \textbf{at}\ \rho \Rightarrow s\}\ ]\!] &&= \overline{\Lambda r.}\ \overline{\lambda x.}\ \mathcal{S}[\![\ s\ ]\!]_{\rho}\\
&\mathcal{E}[\![\ \mathbb{0}\ ]\!] &&= \Lambda a.\ \lambda m.\ m\\
&\mathcal{E}[\![\ e_1 \oplus e_2\ ]\!] &&= \Lambda a.\ \lambda m.\ \mathcal{E}[\![\ e_1\ ]\!]\ a\ (\mathcal{E}[\![\ e_2\ ]\!]\ a\ m)
\end{aligned}$$

Translation of Statements:

$$\begin{aligned}
&\mathcal{S}[\![\ \textbf{val}\ x\ =\ s_0;\ s_1\ ]\!]_{\rho} &&= \lambda k.\ \mathcal{S}[\![\ s_0\ ]\!]_{\rho}\ (\lambda x.\ \mathcal{S}[\![\ s_1\ ]\!]_{\rho}\ k)\\
&\mathcal{S}[\![\ \textbf{return}\ e\ ]\!]_{\rho} &&= \lambda k.\ k\ (\mathcal{E}[\![e]\!])\\
&\mathcal{S}[\![\ e_0[\overline{\rho}](\overline{e})\ ]\!]_{\rho} &&= \mathcal{E}[\![e_0]\!]\ \ \overline{\mathcal{T}[\![\ \rho\ ]\!]}\ \ \overline{\mathcal{E}[\![e]\!]}
\end{aligned}$$

Auxiliary Definitions:

$$\textsc{Cps}\ R\ A\ \ =\ \ (A \to R) \to R$$

Figure 10 Translation from core Λ_ρ to System F.

Together, these corollaries make runtime evidence on the one hand and marker frames on the stack on the other hand redundant. The unwinding can *either* use evidence terms, *or* markers on the stack, since the two agree. The operational semantics uses both to establish this fact. The check for arenas is redundant since it always succeeds. It only exists to establish this fact.

We could erase evidence terms and only rely on marker frames on the stack. In the next section, we are going to CPS where there is no stack. Therefore we will do the opposite: Erase marker frames and purely rely on evidence terms to have the correct content at runtime. This is possible because of the correspondence between evidence and runtime regions. Ultimately, this allows us to prove that CPS translated terms behave exactly as the operational semantics (Theorem 14).

5 Translation of Regions, Pools, and Exceptions to CPS

We now present the translation of Λ_ρ into System F (with file primitives) in CPS. As a result of the translation, the stack K becomes an evaluation context [9], regions become *answer types*, and evidence terms become *answer-type coercions*. As before, we will define the translations of core Λ_ρ and the two extensions with file pools and exceptions step-by-step. Our translation can serve as a compilation technique for languages with control effects and resources into any language that supports first-class functions, making it widely applicable. Moreover, as demonstrated by Schuster et al. [25], modeling control effects with CPS can enable compile-time optimizations for significant performance improvements. We implemented the CPS translation of Λ_ρ as a shallow embedding in Idris 2 [6]. The implementation is attached as supplementary material.

5.1 Translation of Core Λ_ρ

Figure 10 defines the translation of core Λ_ρ to System F. Our translation targets one particular variant of CPS, called *iterated CPS* [10, 24]. Every stack segment, delimited by a marker, is represented by its own continuation argument. That is, in iterated CPS, functions do not

Extended Translation Rules:

$$\begin{array}{lcl}
\mathcal{T}[\![\ \mathsf{Pool}\ \rho\ ]\!] & = & \texttt{PrimPool} \\
\mathcal{T}[\![\ \mathsf{File}\ \rho\ ]\!] & = & \texttt{PrimFile}
\end{array}$$

$$\mathcal{S}[\![\ \textbf{pool}\ \{\ [r](x,\ l) \Rightarrow s_0\ \}\ ]\!]_\rho\ = \\
\quad \textsc{RunPool}\quad (\lambda h.\ (\Lambda r.\ \lambda x.\ \lambda l.\ \mathcal{S}[\![\ s_0\ ]\!]_r)\quad (\mathcal{T}[\![\ \rho\ ]\!])\quad h\quad (\textsc{LiftPool}\ h))$$

$$\begin{array}{lcl}
\mathcal{S}[\![\ \textbf{open}(e,\ e_0,\ i)\ ]\!]_\rho & = & \lambda k.\ k\ (\texttt{openFile}\ \mathcal{E}[\![e]\!]\ \mathcal{E}[\![e_0]\!]) \\
\mathcal{S}[\![\ \textbf{readln}(e,\ i)\ ]\!]_\rho & = & \lambda k.\ k\ (\texttt{readLine}\ \mathcal{E}[\![e]\!])
\end{array}$$

Auxiliary Definitions:

$$\begin{array}{lcl}
\textsc{RunPool} & : & (\texttt{PrimPool} \to \textsc{Cps}\ R\ A) \to \textsc{Cps}\ R\ A \\
\textsc{RunPool} & = & \lambda m.\ \lambda k.\ \textbf{let}\ h\ =\ \texttt{createPool}\ ();\ m\ h\ (\lambda x.\ \texttt{destroyPool}\ h;\ k\ x) \\
\\
\textsc{LiftPool}\ h & : & \forall a.\ \textsc{Cps}\ R\ a \to \textsc{Cps}\ R\ a \\
\textsc{LiftPool}\ h & = & \Lambda a.\ \lambda m.\ \lambda k.\ \texttt{destroyPool}\ h;\ m\ k
\end{array}$$

Figure 11 Translation of Λ_ρ with resource pools.

receive one but potentially multiple continuations. This will only become relevant in the presence of exceptions (Section 5.3).

Translation of Types

Base types, such as Int are left unchanged by the translation. We translate region variables to type variables in System F and the toplevel region to the empty type Void. The translation on types shows that the iterated CPS translation is (so far) very similar to the traditional CPS translation. In particular, the auxiliary meta-definition 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. Evidence terms are functions between effectful computations, as can be seen from the translation of evidence types.

Translation of Terms

As usual in CPS, we translate sequencing of statements to push a frame onto the current continuation k, that is, the continuation first runs s_1 and then continues with k. Return statements are translated to calls to the current continuation. Again, viewing continuations as stacks, this is in accordance with the operational semantics given in Section 4.3. In general, statements with return type τ that have to be run in a region ρ are translated to terms of type Cps $\mathcal{T}[\![\rho]\!]$ $\mathcal{T}[\![\tau]\!]$. This can for instance be seen in the translation of function types. We translate regions to answer types. Region abstractions are translated to type abstractions and region polymorphic functions have a polymorphic answer type. We translate evidence expressions to functions that lift a computation to run in a different region. The reflexivity evidence is translated to the polymorphic identity function, and transitivity of evidence amounts to function composition.

In the remainder of this section, we present the rest of the translation of our language with pools and exceptions Λ_ρ. Later, we show that the translated code in CPS simulates the operational semantics given in Section 4.

5.2 Resource Pools

In Figure 4, we have seen the definition of Λ_ρ with resources pools. Figure 11 defines the translation to CPS. As we have seen in Section 4.7, we do not need any runtime checks to prevent markers and files from being used outside of their region. Indeed, in CPS there is no stack, which we could check for markers.

The **pool** statement creates a fresh resource pool. The translation instantiates r with the outer answer type $\mathcal{T}[\![\ \rho\ ]\!]$. When control leaves the enclosed block, the pool is destroyed. In its translation we use the auxiliary meta function RunPool. It binds the current continuation k and creates a fresh pool h. We run the given computation m with h and a continuation where we push a frame that destroys the pool onto the current continuation k. This ensures that we destroys the pool when we return normally from the enclosed block.

Evidence terms are functions LiftPool h that destroy pool h. Our types make sure that we evaluate the evidence if-and-only-if we non-locally leave the body of the pool. In Section 4.4, evidence was a list of pools. Here, evidence still contains a list of pools, but this list is hidden in the closure environment of the evidence. Evidence composition conceptually concatenates these lists.

The **open** statement opens a file and registers it in the pool. The **readln** statement uses a primitive to read from a file. We require evidence that the pool is live, *i.e.* on the runtime stack, but do not have to actually use it. As we have seen in Section 4.7 its existence is enough to assert that accessing the file is safe.

▶ **Example 10.** Let us consider a simplified version of the motivating example (Section 2.1). The example on the left translates to the term in System F on the right. It has type Cps Void Int.

```
pool {
  [r1](p1: Pool r1, l1: r1 ⊑ T) ⇒
    val f = open(p1, "input", 𝕆);
    return 0
}
```

$$
\begin{array}{l}
\lambda k.\\
\quad \textbf{let}\ h\ =\ \texttt{createPool}\ ();\\
\quad (\Lambda r_1.\ \lambda p_1.\ \lambda l_1.\ \lambda k_1.\\
\qquad \textbf{let}\ f\ =\ \texttt{openFile}\ p_1\ \texttt{"input"};\\
\qquad k_1\ 0)\ \mathsf{Void}\ h\\
\qquad\quad (\Lambda a.\ \lambda m.\ \lambda k.\ \texttt{destroyPool}\ h;\ m\ k)\\
\qquad\quad (\lambda x.\ \texttt{destroyPool}\ h;\ k\ x)
\end{array}
$$

This term can be normalized to the following:

$$\lambda k.\ \textbf{let}\ h\ =\ \texttt{createPool}\ ();\ \textbf{let}\ f\ =\ \texttt{openFile}\ h\ \texttt{"input"};\ \texttt{destroyPool}\ h;\ k\ 0$$

5.3 Exceptions

In this subsection, we present the translation of exceptions. Whereas in the operational semantics (Section 4.5) we have divided the stack into regions with markers, we now have multiple stacks, *i.e.* continuations. We have seen that evidence terms contained exactly the list of markers we have to unwind when we throw to a handler. Now we take advantage of this fact and let the evidence be the unwinding action itself. Figure 12 presents the translation of exceptions. It is different from the translation to double-barreled CPS from Kennedy [16], where functions only ever get exactly two continuations. Under our translation to iterated CPS functions can receive any number of continuations.

To support aborting the computation, we instantiate the answer type r of the translated body s_0 to be the type Cps $\mathcal{T}[\![\rho]\!]\ \mathcal{T}[\![\tau]\!]$. This adds another layer of CPS and one additional (curried) continuation argument. In the translation of the **try** ... **catch** ... statement, we use

Extended Translation Rules:

$$\mathcal{T}[\![\ \mathsf{Catch}\ \rho\ ]\!] = \textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \mathsf{Void}$$

$$\mathcal{S}[\![\ \textbf{try}\ \{\ [r](x,\ l) \Rightarrow s_0\ \}\ \textbf{catch}\ \{\ s\ \}\ ]\!]_\rho = \textsc{RunCps}\ ((\Lambda r.\ \lambda x.\ \lambda l.\ \mathcal{S}[\![\ s_0\ ]\!]_r)\ (\textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \mathcal{T}[\![\ \tau\ ]\!])\ \ (\lambda k.\ \mathcal{S}[\![\ s\ ]\!]_\rho)\ \ (\textsc{LiftCps}))$$

$$\mathcal{S}[\![\ \textbf{throw}(e,\ i)\ ]\!]_\rho = \mathcal{E}[\![i]\!]\ \mathsf{Void}\ \mathcal{E}[\![e]\!]$$

Auxiliary Definitions:

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

Figure 12 Translation of Λ_ρ with exceptions.

RunCps. It runs the given computation m with an additional continuation which is initially empty. The evidence l lifts the given computation from the inner region to the outer region. It will be bound to LiftCps which pushes the current continuation onto the next one.

A Catch ρ is a CPS expression that aborts the computation. That is, the handler $(\lambda k.\ \mathcal{S}[\![\ s\ ]\!]_\rho)$ discards the current continuation k. In the translation of statement **throw**$(e,\ i)$, we call the provided evidence i and then the handler e. Running the evidence lifts the handler into the correct region, making it compatible with the current answer type. It is safe for the handler to discard the continuation k, since all cleanup actions contained in k are run by the evidence.

▶ **Example 11.** Let us consider the example from Section 2.2. The example on the left translates to the resulting term of type Cps Void Int on the right.

```
try { [r1](e1 : Catch r1, l1 : r1 ⊑ T) ⇒
  safeDiv[r1](5, 0, e1)
} catch {
  return 0
}
```

$$\begin{array}{l} (\Lambda r_1.\ \lambda e_1.\ \lambda l_1. \\ \quad \texttt{safeDiv}\ r_1\ 5\ 0\ e_1 \\ )\ (\textsc{Cps}\ \mathsf{Void}\ \mathsf{Int}) \\ \quad (\lambda k_1.\ \lambda k_2.\ k_2\ 0) \\ \quad (\Lambda a.\ \lambda m.\ \lambda k.\ \lambda j.\ m\ (\lambda x.\ k\ x\ j)) \\ \quad (\lambda x.\ \lambda k.\ k\ x) \end{array}$$

The resulting System F term can be beta reduced to:

$$\lambda k_2.\ \texttt{safeDiv}\ (\textsc{Cps}\ \mathsf{Void}\ \mathsf{Int})\ 5\ 0\ (\lambda k_1.\ \lambda k_2.\ k_2\ 0)\ (\lambda x.\ \lambda k.\ k\ x)\ k_2$$

We instantiate the answer type r of `safeDiv` with r_1, which itself is instantiated to Cps Void Int, adding one layer of continuations. The return type is Cps (Cps Void Int) Int and our program receives two continuations. To abort, the exception handler discards the first (*i.e.*, k_1) and returns 0 to the second (k_2).

5.4 Combining Resource Pools and Exceptions

First of all, let us briefly note that we translate well-typed programs in full Λ_ρ to well-typed programs in System F.

▶ **Theorem 12** (Well-typedness of Translated Terms)**.**
If $\Gamma \shortmid \rho \vdash\ s\ :\ \tau$, *then* $\mathcal{T}[\![\ \Gamma\ ]\!] \vdash\ \mathcal{S}[\![s]\!]_\rho\ :\ (\mathcal{T}[\![\ \tau\ ]\!] \to \mathcal{T}[\![\ \rho\ ]\!]) \to \mathcal{T}[\![\ \rho\ ]\!]$

Proof. Straightforward induction over the typing derivation. ◀

The translation of exception handlers in Section 5.3 automatically interacts correctly with the evidence terms we have defined for resource pools in Section 5.2: We clear a pool exactly when an exception is thrown across it. This is because we have chosen the translation of evidence to be a concrete computation that moves from one region to another one.

▶ **Example 13.** The following is an extended example where we combine resource pools and exceptions in a more complicated way. The program splits a large input file into smaller files of 100 lines each.

```
try { [r1](stop : Catch r1, l1 : r1 ⊑ Top) ⇒
  withFile[r1]("input", { [r2](in: File r2, l2 : r2 ⊑ r1) ⇒
    def copyFile(target : String) at r2 {
      withFile[r2](target, { [r3](out: File r3, l3 : r3 ⊑ r2) ⇒
        def copyLine() at r3 {
          if (isEOF(in, l3)) { throw(stop, l3 ⊕ l2) }
          else { writeln(out, readln(in, l3), 𝟘) }
        };
        def innerLoop(toCopy : Int) at r3 {
          if (toCopy > 0) { copyLine(); innerLoop(toCopy - 1) }
        };
        innerLoop(100)
      })
    };
    def loop(n : Int) at r2 { copyFile("output" ++ n); loop(n + 1) };
    loop(0)
  })
} catch { return () }
```

When we encounter the end of the input file, we simply throw an exception to terminate the program. We can be confident that all resources will be properly cleaned up and so fearlessly use exceptions to structure control flow. The outer loop, for example never returns. It is terminated by throwing an exception. This program, after CPS translation, manually applying contification [16], and beta reduction, reduces to the code in Figure 13.

Our CPS translation of both regions and control enables aggressive optimization. For example, at the end of the input file, we immediately destroy both pools and return. Since we only apply well-known optimizations on functional programs, we can be certain of their correctness without having to reason explicitly about resources nor control effects nor their combination. The overall correctness of the optimized result rests on the correctness of our CPS translation.

5.5 Simulation of the Operational Semantics by the CPS translation

In Section 4, we defined an operational semantics for Λ_ρ. In this section we defined a CPS translation for Λ_ρ. We now show that the two behave the same. This entails that the operational properties from Section 4 carry over to the CPS translation and that optimization via beta reduction is sound. To show preservation of semantics, we translate statements to terms and stacks to evaluation contexts in System F. We define the translation $\mathcal{M}[\![\cdot]\!]$ of machine states as the plugging of the translation of the statement into the translation of the stack. The translation given in an Appendix which is attached as supplementary material.

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

```
λk.
  let p2 = createPool ();
  let in = openFile p2 "input";
  let rec loop n = (λk1.
    let p3 = createPool ();
    let out = openFile p3 ("output" ++ n);
    let rec innerLoop toCopy = (λk2.
      if (toCopy > 0)
      then if isEOF(in)
        then destroyPool p3; destroyPool p2; (λk4. k4 0)
        else let line = readLine in; writeLine out line; innerLoop (toCopy − 1)
      else destroyPool p3; loop (n + 1)
    );
    innerLoop 100
  );
  loop 0 k
```

Figure 13 Result of translating Example 13 to CPS.

▶ **Theorem 14** (Simulation).
If $M \to M'$*, then* $\mathcal{M}[\![\ M\ ]\!] \to^* \mathcal{M}[\![\ M'\ ]\!]$.

Proof. By considering each case of the stepping relation. The *(throw)* step needs its own lemma, which we show by induction on possible evidence expressions. ◀

Since for simulation we are only interested in operational behavior, we target the untyped lambda calculus (with primitives for file management) instead of System F. The translation of statements is the same as $\mathcal{S}[\![\ s\ ]\!]_\rho$ in Figures 10, 11, and 12, but we erase all type annotations, type abstractions, and type applications. There is no harm in doing so, since our target is in CPS where the evaluation order is explicit.

While the operational semantics given in Section 4 discards frames during unwinding, for our proof of simulation we have to retain them. We do so in a third component of the machine state $\langle\ \mathbf{throw}(h,\ w)\ \|\ \mathsf{K}\ \|\ \mathsf{H}\ \rangle$: the stack trace H. This is necessary because the CPS translation discards the whole continuation in one step, while the operational semantics slowly unwinds the stack frame-by-frame.

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.

▶ **Example 15.** Pools are created and destroyed exactly when they would be in the operational semantics. As an illustration, consider the following sequence of machine steps where we unwind a pool frame:

$$\langle\ \mathbf{throw}(h_1,\ (\mathbf{po}\ h_2\ ::\ \bullet))\ \|\ \#\mathbf{pool}_{h_2}\ \{\ \square\ \}\ ::\ \#\mathbf{catch}_{h_1}\ \{\ \square\ \}\ \{\ \mathbf{return}\ 1\ \}\ ::\ \bullet\ \rangle \to$$

$$\langle\ \mathbf{throw}(h_1,\ (\mathbf{po}\ h_2\ ::\ \bullet))\ \|\ \#\mathbf{pool}_{h_2}\ \{\ \square\ \}\ ::\ \#\mathbf{catch}_{h_1}\ \{\ \square\ \}\ \{\ \mathbf{return}\ 1\ \}\ ::\ \bullet\ \|\ \bullet\ \rangle \to$$

$$\langle\ \mathbf{throw}(h_1,\ \bullet)\ \|\ \#\mathbf{catch}_{h_1}\ \{\ \square\ \}\ \{\ \mathbf{return}\ 1\ \}\ ::\ \bullet\ \|\ \#\mathbf{pool}_{h_2}\ \{\ \square\ \}\ ::\ \bullet\ \rangle \to$$

$$\langle\ \mathbf{return}\ 1\ \|\ \bullet\ \rangle$$

The first step *(throw)* goes from normal execution to the unwinding state which accumulates frames in its third component. The next two steps are *(free)* and *(catch)*. In CPS, we can observe the same program trace:

$$((\textsc{LiftPool}\ h_2)\ (\lambda k_1.\ \lambda k_2.\ k_2\ 1))\ (\lambda x.\ \texttt{destroyPool}\ h_2;\ (\lambda x.\ \lambda k.\ k\ x)\ x)\ \texttt{done} \to$$
$$(\lambda k.\ \texttt{destroyPool}\ h_2;\ (\lambda k_1.\ \lambda k_2.\ k_2\ 1)\ k)\ (\lambda x.\ \texttt{destroyPool}\ h_2;\ (\lambda x.\ \lambda k.\ k\ x)\ x)\ \texttt{done} \to$$
$$(\lambda k_1.\ \lambda k_2.\ k_2\ 1)\ (\lambda x.\ \texttt{destroyPool}\ h_2;\ (\lambda x.\ \lambda k.\ k\ x)\ x)\ \texttt{done} \to$$
$$(\lambda k_2.\ k_2\ 1)\ \texttt{done}$$

▶ **Example 16.** Although we do not have any markers generated at runtime, the CPS translation exactly mimics the behavior of the operational semantics, which does have them. Consider another example, where we throw an exception to an outer handler. The steps are *(throw)*, *(forward)*, and *(catch)*.

$$\langle\ \textsf{throw}(h_1,\ (h_2\ ::\ \bullet))\ \|\ \#\textsf{catch}_{h_2}\ \{\ \square\ \}\ \{\ \textsf{return}\ 2\ \}\ ::\ \#\textsf{catch}_{h_1}\ \{\ \square\ \}\ \{\ \textsf{return}\ 1\ \}\ ::\ \bullet\ \rangle \to$$
$$\langle\ \textsf{throw}(h_1,\ (h_2\ ::\ \bullet))\ \|\ \#\textsf{catch}_{h_2}\ \{\ \square\ \}\ \{\ \textsf{return}\ 2\ \}\ ::\ \#\textsf{catch}_{h_1}\ \{\ \square\ \}\ \{\ \textsf{return}\ 1\ \}\ ::\ \bullet\ \|\ \bullet\ \rangle \to$$
$$\langle\ \textsf{throw}(h_1,\ \bullet)\ \|\ \#\textsf{catch}_{h_1}\ \{\ \square\ \}\ \{\ \textsf{return}\ 1\ \}\ ::\ \bullet\ \|\ \#\textsf{catch}_{h_2}\ \{\ \square\ \}\ \{\ \textsf{return}\ 2\ \}\ ::\ \bullet\ \rangle \to$$
$$\langle\ \textsf{return}\ 1\ \|\ \bullet\ \rangle$$

In CPS, we start out with three continuations, then we push the first one onto the second one, then the exception handler discards both in one step:

$$(\textsc{LiftCps}\ (\lambda k_1.\ \lambda k_2.\ k_2\ 1))\ (\lambda x.\ \lambda k.\ k\ x)\ (\lambda x.\ \lambda k.\ k\ x)\ \texttt{done} \to$$
$$(\lambda k.\ \lambda j.\ (\lambda k_1.\ \lambda k_2.\ k_2\ 1)\ (\lambda y.\ k\ y\ j))\ (\lambda x.\ \lambda k.\ k\ x)\ (\lambda x.\ \lambda k.\ k\ x)\ \texttt{done} \to$$
$$(\lambda k_1.\ \lambda k_2.\ k_2\ 1)\ (\lambda y.\ (\lambda x.\ \lambda k.\ k\ x)\ y\ (\lambda x.\ \lambda k.\ k\ x))\ \texttt{done} \to$$
$$(\lambda k_2.\ k_2\ 1)\ \texttt{done}$$

The CPS translation exhibits the same behavior as the operational semantics. It simulates the generative semantics of exceptions. Remarkably, it does not need any runtime support for markers on the stack to do so. Indeed, in CPS there is no stack!

6 Related Work

Out of the huge body of work on regions, the one most closely related, and indeed which has been the basis of our work, is [18], which in turn is based on [11]. Kiselyov and Shan provide a library for region-based resource management in Haskell. They demonstrate how types, regions, and subregioning evidence are inferred, which we do not discuss. They deal with builtin Haskell exceptions, but leave a formal proof to future work. We go further, and add exceptions as a language feature, and prove region- and exception safety. Moreover, we present a CPS translation of these features.

Crary et al. [8] present a language with *dynamic* regions, where regions do not have to be nested, resource access is safe, but resource cleanup is not automatic but explicit. Their language is presented in CPS. Indeed, to quote Fluet et al. [12]: "Dynamic regions are not restricted to LIFO lifetimes and can be treated as first-class objects. They are particularly well suited for iterative computations, CPS-based computations, and event-based servers where lexical regions do not suffice." We present a CPS translation of *lexical* regions where resources are automatically destroyed, even when an exception is thrown.

Clearly also related is the line of work on monadic encapsulation of state [19, 22, 26]. The most recent work in this line [27] presents a mechanized proof of a number of equivalences in the presence of encapsulated mutable state. We merely prove that references are not used outside of their region, but do so in the presence of exceptions.

Our CPS translation of exceptions is closely related to the one presented by Schuster et al. [25]. However, they do not support effect-polymorphic functions. Our translation to System F is similar to the one for effect handlers sketched in Appendix B of [14].

Kiselyov and Ishii [17] present a Haskell library for effect handlers based on a variant of the free monad in Haskell. Their library supports user-defined effects and handlers and they provide a range of pre-defined effects like exceptions, non-determinism, and state. They also discuss a region effect for safe and automatic allocation and disposal of resources, which correctly works in the presence of the exception effect. Other effects, like non-determinism, are explicitly ruled out by the type system when they would be used across a resource delimiter. They reify the structure of the program as a free monad and then write interpreters over this structure, whereas we translate programs to CPS. Moreover we provide a proof of region- and exception safety, which is out of scope of their work.

Leijen [20] reports on an extension of the programming language Koka with support for resources and finalization. They support general effect handlers, while we merely discuss the special case of exceptions. Their approach requires sophisticated modification of the language runtime, whereas our approach can be explained as a translation to pure System F. They allow for more complex finalization patterns, where users explicitly run the finalizers of a resumption. This is to avoid running finalizers on linearly used resumptions, a problem that we completely side-step by only discussing exceptions.

Ahman and Bauer [1] present an approach to resources management: Runners. They guarantee that cleanup actions are run exactly once. We offer the same guarantee. We present an operational semantics that relates resource management to the stack and a translation of programs to CPS. Their denotational semantics translates programs to essentially a free monad.

7 Conclusion

We presented Λ_ρ, a language with first-class function, regions, resources, and exceptions. Its type system guarantees safe access to resources and safe use of exceptions. We then presented a CPS translation that preserves these guarantees.

We view regions as describing runtime stacks. This view is very much in line with recent work on effect handlers. One does wonder if our approach scales to more general control effects, which do not discard the current continuation, and perhaps even uses it multiple times. This is the subject of ongoing investigation.

References

1 D. Ahman and A. Bauer. Runners in action. In P. Müller, editor, *Programming Languages and Systems*, pages 29–55, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44914-8.

2 A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.

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

4 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.

5 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. URL `https://doi.org/10.1145/3428194`.

6 E. Brady. Idris 2: Quantitative type theory in action. Technical report, University of St Andrews, Scotland, UK, 2020. URL https://www.type-driven.org.uk/edwinb/papers/idris2.pdf.

7 Y. Cong, L. Osvald, G. M. Essertel, and T. 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. URL http://doi.acm.org/10.1145/3341643.

8 K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 262–275, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130953. doi: 10.1145/292540.292564. URL https://doi.org/10.1145/292540.292564.

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

10 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.

11 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. URL https://doi.org/10.1145/1016850.1016867.

12 M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In P. Sestoft, editor, *Programming Languages and Systems*, pages 7–21, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33096-7.

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. URL https://doi.org/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 A. Kennedy. Compiling with continuations, continued. In *Proceedings of the International Conference on Functional Programming*, pages 177–190, New York, NY, USA, 2007. ACM.

17 O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Proceedings of the Haskell Symposium*, pages 94–105, New York, NY, USA, 2015. ACM.

18 O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *Proceedings of the Haskell Symposium*, Haskell '08, New York, NY, USA, 2008. ACM.

19 J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 24–35, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 089791662X. doi: 10.1145/178243.178246. URL https://doi.org/10.1145/178243.178246.

20 D. Leijen. Algebraic effect handlers with resources and deep finalization. Technical Report MSR-TR-2018-10, Microsoft Research, April 2018.

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

22 E. Moggi and A. Sabry. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming*, 11(6):591–627, Nov. 2001.

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 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.

25 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. URL `https://doi.org/10.1145/3408975`.

26 M. Semmelroth and A. Sabry. Monadic encapsulation in ml. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, page 8–17, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131119. doi: 10.1145/317636.317777. URL `https://doi.org/10.1145/317636.317777`.

27 A. Timany, L. Stefanesco, M. Krogh-Jespersen, and L. Birkedal. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runst. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017. doi: 10.1145/3158152. URL `https://doi.org/10.1145/3158152`.

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. URL `https://doi.org/10.1006/inco.1996.2613`.

29 M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, and P. Sestoft. Programming with regions in the ml kit (for version 4). 10 2001.

30 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. URL `https://doi.org/10.1145/3408981`.

31 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.

32 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.

Syntax of Lambda Calculus:

Terms
$$t \quad ::= \quad x \mid \lambda x.\ t \mid t\ t$$

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

Plugging
$$\begin{array}{lcl} plug(t,\ \bullet) & = & t \\ plug(t,\ \Box\ v\ ::\ C) & = & plug(t\ v,\ C) \\ plug(t,\ \textsf{let}\ \kappa\ =\ v\ \textsf{in}\ \Box\ ::\ C) & = & plug(t\ [\kappa \mapsto v],\ C) \end{array}$$

Syntax of Stack Traces:

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

Syntax of Machine States:

$$\begin{array}{lcll} \mathsf{M} & ::= & \langle\ s \parallel \mathsf{K}\ \rangle & \text{execution} \\ & \mid & \langle\ \textsf{throw}(h,\ w) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle & \text{unwinding} \end{array}$$

Translation of Machine States:

$$\begin{array}{lcl} \mathcal{M}[\![\ \langle\ s \parallel \mathsf{K}\ \rangle\ ]\!] & = & plug(\mathcal{S}[\![\ s\ ]\!]\ ,\ \Box\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) \\ \mathcal{M}[\![\ \langle\ \textsf{throw}(h,\ w) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle\ ]\!] & = & plug(\mathcal{W}[\![\ w\ ]\!]_{\mathcal{E}[\![\ h\ ]\!]}\ ,\ \Box\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) \end{array}$$

Translation of Stacks:

$$\begin{array}{lcl} \mathcal{K}[\![\ \bullet\ ]\!] & = & \textsf{let}\ \kappa\ =\ \texttt{done}\ \textsf{in}\ \Box\ ::\ \bullet \\ \mathcal{K}[\![\ \textsf{val}\ x\ =\ \Box;\ s\ ::\ \mathsf{K}\ ]\!] & = & \textsf{let}\ \kappa\ =\ \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa\ \textsf{in}\ \Box\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!] \\ \mathcal{K}[\![\ \#\textsf{pool}_h\ \{\ \Box\ \}\ ::\ \mathsf{K}\ ]\!] & = & \textsf{let}\ \kappa\ =\ \lambda x.\ \texttt{destroyPool}\ h;\ \kappa\ x\ \textsf{in}\ \Box\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!] \\ \mathcal{K}[\![\ \#\textsf{catch}_h\ \{\ \Box\ \}\ \{\ s\ \}\ ::\ \mathsf{K}\ ]\!] & = & \textsf{let}\ \kappa\ =\ \lambda x.\ \lambda k.\ k\ x\ \textsf{in}\ \Box\ ::\ \Box\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!] \end{array}$$

Translation of Stack Traces:

$$\begin{array}{lcl} \mathcal{H}[\![\ \bullet\ ]\!] & = & \kappa \\ \mathcal{H}[\![\ \textsf{val}\ x\ =\ \Box;\ s\ ::\ \mathsf{H}\ ]\!] & = & \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa] \\ \mathcal{H}[\![\ \#\textsf{pool}_h\ \{\ \Box\ \}\ ::\ \mathsf{H}\ ]\!] & = & \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \lambda x.\ \texttt{destroyPool}\ h;\ \kappa\ x] \\ \mathcal{H}[\![\ \#\textsf{catch}_h\ \{\ \Box\ \}\ \{\ s\ \}\ ::\ \mathsf{H}]\!] & = & \lambda y.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \lambda x.\ \lambda k.\ k\ x]\ y\ \kappa \end{array}$$

Translation of Unwinding:

$$\begin{array}{lcl} \mathcal{W}[\![\ \bullet\ ]\!]_t & = & t \\ \mathcal{W}[\![\ \textsf{po}\ h\ ::\ w\ ]\!]_t & = & \lambda k.\ \texttt{destroyPool}\ h;\ \mathcal{W}[\![\ w\ ]\!]_t\ k \\ \mathcal{W}[\![\ \textsf{ca}\ h\ ::\ w\ ]\!]_t & = & \lambda k.\ \lambda j.\ \mathcal{W}[\![\ w\ ]\!]_t\ (\lambda x.\ k\ x\ j) \end{array}$$

Translation of Evidence values:

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

Translation of Handlers:

$$\mathcal{E}[\![\ h\ ]\!] \quad = \quad \lambda k.\ \mathcal{S}[\![\ s\ ]\!] \quad \text{where } \#\textsf{catch}_h\ \{\ \Box\ \}\ \{\ s\ \}\ in\ \mathsf{K}$$

Figure 14 Translation of machine states.

A Abstract Machine and Simulation

A.1 Translation of Machine States

Figure 14 lists the translation of machine states to untyped lambda calculus. 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, $\textbf{let}\ \kappa\ =\ v\ \textbf{in}\ \square$). Plugging a term into a context is straight-forward. The case of the continuation binder performs substitution.

Translation of Machine States

The translation of execution machine states (that is, $\langle\ s\ \|\|\ \mathsf{K}\ \rangle$) 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, $\langle\ \textbf{throw}(h,\ w)\ \|\|\ \mathsf{K}\ \|\|\ \mathsf{H}\ \rangle$) plugs the unwinding term (that is, $\mathcal{W}[\![\ w\ ]\!]_{\mathcal{E}[\![\ h\ ]\!]}$) into an application to the translation of the stack trace ($\square\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]$), and then into the translation of the stack ($\mathcal{K}[\![\ \mathsf{K}\ ]\!]$). 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.

Proof of Simulation

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

▶ **Lemma 17.** $\mathcal{S}[\![\ s\ ]\!]\ [x \mapsto \mathcal{E}[\![\ e\ ]\!]\ ]\ =\ \mathcal{S}[\![\ s\ [x \mapsto e]\ ]\!]$

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

▶ **Lemma 18** (Unwinding). $\mathcal{E}[\![\ i\ ]\!]\ t \to^*\ \mathcal{W}[\![\ \mathcal{V}[\![\ i\ ]\!]\ ]\!]_t$

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

Given the above lemmas, we can show simulation:

Proof of Theorem 14 (Simulation). The proof proceeds by case analysis of the step taken by the abstract machine. Most of them are straight-forward, except for *(throw)*, which requires Lemma 18:

$$
\begin{array}{lcl}
\mathcal{M}[\![\ \langle\ \textbf{throw}(h,\ i)\ \|\|\ \mathsf{K}\ \rangle\ ]\!] & = & \text{by Definition } \mathcal{M}[\![\ \cdot\ ]\!] \\
\mathit{plug}(\mathcal{S}[\![\ \textbf{throw}(h,\ i)\ ]\!],\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = & \text{by Definition } \mathcal{S}[\![\ \cdot\ ]\!] \\
\mathit{plug}(\mathcal{E}[\![\ i\ ]\!]\ \mathcal{E}[\![\ h\ ]\!],\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & \to^* & \text{by Lemma Unwinding} \\
\mathit{plug}(\mathcal{W}[\![\ \mathcal{V}[\![\ i\ ]\!]\ ]\!]_{\mathcal{E}[\![\ h\ ]\!]},\ \square\ \kappa\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = & \text{by Definition } \mathcal{H}[\![\ \cdot\ ]\!] \\
\mathit{plug}(\mathcal{W}[\![\ \mathcal{V}[\![\ i\ ]\!]\ ]\!]_{\mathcal{E}[\![\ h\ ]\!]},\ \square\ \mathcal{H}[\![\ \bullet\ ]\!]\ ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = & \text{by Definition } \mathcal{M}[\![\ \cdot\ ]\!] \\
\mathcal{M}[\![\ \langle\ \textbf{throw}(h,\ \mathcal{V}[\![\ i\ ]\!])\ \|\|\ \mathsf{K}\ \|\|\ \bullet\ \rangle\ ]\!] & &
\end{array}
$$

◀

Stack Typing:

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

$$\frac{}{\vdash \bullet : \mathsf{Int}}\ [\textsc{Exit}]$$

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

$$\frac{\vdash \mathsf{K} : \tau}{\vdash \#\mathbf{pool}_h\ \{\ \square\ \} :: \mathsf{K} : \tau}\ [\#\textsc{Pool}]$$

$$\frac{\emptyset \mid \mathcal{R}[\![\, \mathsf{K} \,]\!] \vdash s : \tau \qquad \vdash \mathsf{K} : \tau}{\vdash \#\mathbf{catch}_h\ \{\ \square\ \}\ \{\ s\ \} :: \mathsf{K} : \tau}\ [\#\textsc{Catch}]$$

Figure 15 Stack typing of Λ_ρ

A.2 Abstract Machine Typing

Showing Preservation (Theorem 5) requires typing of stacks. Figure 15 lists the corresponding typing rules.