All About That Stack

A Unified Treatment of Regions and Control Effects

PHILIPP SCHUSTER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, EPFL, Switzerland
KLAUS OSTERMANN, University of Tübingen, Germany

Ever since the inception of Algol have programming language researchers sought good abstractions to inspect and manipulate stacks while maintaining basic invariants of program behavior. These abstractions range from procedure calls and block structure to region-based resource management and control effects. While all these abstractions are useful and well-designed individually, their combination and interaction is an open issue. We present a conceptual framework with a novel form of stack abstraction, in which stacks are decomposed into regions, moves between stacks are expressed as control effects, and relationships between regions are represented with subregioning evidence. We demonstrate and prove that these abstractions are powerful enough to express and combine region-based resource management and control effect while guaranteeing region and effect safety invariants. We also discuss an implementation by means of a compilation to System F and validate its utility by means of several standard examples.

1 INTRODUCTION

Regions are a useful concept in programming languages for the safe and automatic management of resources [Tofte and Talpin 1997]. Resources are organized into a stack of regions and automatically released when control flow leaves the part of the program where a region is live. Control effects, like for example exceptions or more general control operators (such as shift / reset [Danvy and Filinski 1990] or algebraic effect handlers [Plotkin and Pretnar 2013]), present a challenge for region-based resource management, because they allow for non-local transfer of control. While some work [Grossman et al. 2002; Kiselyov and Ishii 2015; Kiselyov and Shan 2008; Tofte et al. 2001] mentions compatibility with exceptions, a formal argument for the correct interaction between region-based resource management and control effects is rarely given.

In this paper we present a conceptual framework that uniformly accommodates existing use-cases of regions as well as different control effects. Our framework does not resolve all problematic interactions between resource management and control effects, but it serves as a tool to uniformly reason about both domains.

Our unified treatment is *"all about that stack"*. We understand *regions*, *control effects*, and subregioning *evidence* by their connection to the runtime stack, as illustrated in Figure 1, which we discuss in detail in the following section.

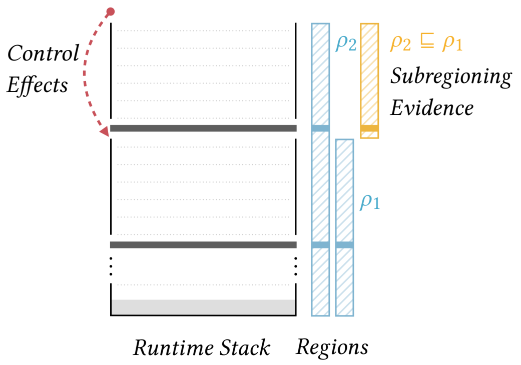

Fig. 1. Illustration of the core concepts *regions of the stack* (*i.e.*, ρ_1 and ρ_2), control flow transfer via *control effects*, and *evidence* (*i.e.*, $\rho_2 \sqsubseteq \rho_1$) between regions.

One example for the interaction between resources and control effects are finalizers. What should happen when finalization itself throws an exception during unwinding? The behavior varies among programming languages. Our framework helps us to not only discuss *which* semantics is appropriate in this case, but also to argue *why* this behavior is safe.

1.1 Overview

The rest of the paper is organized as follows. In Section 2, we introduce the main ideas behind our conceptual framework by studying two different language features and their interaction: arena based memory-management and exceptions. In Section 3, we present a base language Λ_ρ with type-level region tracking and term-level subregioning evidence. The formulation of this base calculus is parametrized over the semantic interpretation of both *regions* and *subregioning evidence*. We extend this base language with arena-based memory management and exceptions. We then present an operational semantics that formally establishes the connection between type-level regions and the concrete runtime stack during execution. In Section 4, we define a denotational semantics for the same language as a translation to System F in continuation-passing style. To evaluate the applicability of our conceptual framework, in subsequent sections we extend the base calculus Λ_ρ with more interesting control constructs, like operators for delimited control and effect handlers, as well as more interesting constructs for dealing with resources, like backtrackable mutable state and dynamic wind. Our unified treatment of effects and regions puts us in a unique position to discuss these features and their interaction in the same language.

1.2 Contributions

In particular, this paper makes the following contributions:

- A conceptual framework of regions and subregioning evidence, allowing us to perform an in-depth study of the interaction of region-based resource management and control effects in a type- and effect-safe language Λ_ρ. Soundness proofs of Λ_ρ are mechanized in Coq.
- An operational semantics and theorems of correspondence that connect type-level regions and term-level subregioning evidence with run-time properties during evaluation (Corollaries 3.3 and 3.4). Together these theorems entail that we do not need special language runtime support for many region-related language features.
- A denotational semantics of Λ_ρ in terms of an iterated CPS translation where regions are *answer types* and evidence terms are *answer-type coercions*. We present several case studies that instantiate our conceptual framework and discuss non-trivial interactions between different region-related language features. We specify the semantics of those features without adding special runtime support or changing the denotation of regions or evidence.
- The translations to CPS have been implemented as a shallow embedding into the dependently-typed language Idris, which shows that they take well-typed terms to well-typed terms in System F. All examples given in the paper type check and evaluate to the expected result.

2 MAIN IDEAS

As already mentioned, our unified treatment is *"all about that stack"* and that we understand *regions*, *control effects*, and subregioning *evidence* by their connection to the runtime stack, as illustrated in Figure 1. Let us consider each of the three concepts in turn.

Regions. At the heart of this unified treatment lies our understanding of what a region is. We shift the perspective and instead of considering a *stack of regions* [Tofte and Talpin 1994], we consider *regions of the stack*. That is, where most literature on region-based memory management sees a region as a part of the store (*i.e.*, a "region of memory"), we understand a region as part of the runtime stack. While other approaches track where values are stored, we rather track where computations are run. Importantly, in this paper with *stack* we refer to the runtime stack and *not* to a way of organizing memory.

Control Effects. While regions denote particular parts of the runtime stack, control effects (like exceptions) *move between stack segments*. As we will see in this section, unifying the treatment of regions and control effects, we rephrase the problem of effect safety as a problem of region safety, and use the same type-level machinery to guarantee both. This understanding of effect safety is very much in line with recent work on effect handlers [Biernacki et al. 2019a; Brachthäuser et al. 2020a; Xie et al. 2020; Zhang and Myers 2019].

Subregioning Evidence. Since they denote parts of the runtime stack, regions are naturally nested. To witness this nesting, we introduce explicit term-level evidence [Fluet and Morrisett 2004]. There are two important aspects to our notion of subregioning evidence, corresponding to their *static* and *dynamic* interpretation. Statically, evidence of type $\rho_2 \sqsubseteq \rho_1$ witnesses the fact that region ρ_2 is nested within region ρ_1, allowing us to guarantee region and effect safety. Importantly, evidence also has a *runtime interpretation*: dynamically, evidence of the type $\rho_2 \sqsubseteq \rho_1$ denotes the *difference* between the two regions ρ_2 and ρ_1. We assign meaning to this difference and equip evidence with computational content describing what it means for program execution to move from one region of the stack to another region of the stack. In their work on monadic regions, Fluet and Morrisett [2004] incorporate a very similar form of subregioning, envisioning that

[...] we can imagine a scheme in which this primitive evidence is abstract and we provide additional operations for combining evidence [...] – Fluet and Morrisett [2004, p. 106]

In this paper, we do exactly that. In the remainder, we will encounter different semantic interpretations of this "difference". The following subsections use arena allocation and exceptions as examples to make these ideas more concrete.

2.1 Arena-based Memory Management

As a first example, let us see how region-based resource management can be expressed within our framework. We use memory management as an example. Resources other than memory, for example file handles [Kiselyov and Shan 2008], would be treated similarly. Our type system follows Fluet and Morrisett [2004]. What is new is our understanding what a region *is*.

An arena is a block of memory that allows for the allocation of many small objects into it. While we use the term arena, other terminology such as *pool* and *region* is also in use. In alignment with our understanding of regions, arenas are tied to the runtime stack: they have to be allocated and deallocated in a last-in-first-out way.

To make arena-based memory management safe, we have to ensure that we only allocate into an arena while it is still *live* and that we only read from a pointer into an arena which is still live. An arena is live, when it is in the region which describes the current runtime stack. To understand this intuitively, consider the following example.

Example 2.1. In this example, we create a fresh arena. Operationally, the `arena { ... }` statement will allocate a fresh arena, and deallocate it after control flow leaves the enclosed block. It introduces a *region variable* $\mathtt{r}_1$, an *arena* $\mathtt{a}_1$ and *subregioning evidence* $\mathtt{l}_1$. In our type system, every statement is checked in a region. The enclosed block is checked in region $\mathtt{r}_1$.

```
arena { [r₁](a₁ : Arena r₁ A, l₁ : r₁ ⊑ ⊤) ⇒
  val ptr = alloc(a₁, aValue, 𝟘);
  arena { [r₂](a₂ : Arena r₂ (A × A), l₂ : r₂ ⊑ r₁) ⇒
    val pair = alloc(a₂, (load(ptr, l₂), load(ptr, l₂)), 𝟘);
    return pair // does not type check
  }
}
```

We then allocate a value aValue into the arena a_1. To allocate into the arena a_1, we have to provide evidence that the arena's region is nested inside of the current region, *i.e.* that $r_1 \sqsubseteq r_1$. We provide the reflexivity evidence $\mathbb{0}$. The resulting pointer ptr has type $\text{Ptr}\ r_1\ A$.

We then create a second arena a_2 in a second region r_2 which is clearly inside of r_1. This fact is witnessed by the evidence variable l_2. We allocate a pair into a_2. To load from the pointer ptr, we have to provide evidence that the pointer's region is inside of the current region, *i.e.* that $r_2 \sqsubseteq r_1$. We provide the evidence variable l_2. The allocated pair pair has type $\text{Ptr}\ r_2\ (A \times A)$. This pointer shall not be used outside of region r_2. Our type system prevents pair from being returned from the block. At the same, it would be fine to return ptr from the inner region r_2, but not from the outer region r_1.

Meaning of Regions and Evidence. In the case of arenas, a region is a concrete *list of live arenas*. The top-level region is the empty list. When we run this example, we allocate a fresh arena a_1. Region r_1 stands for the singleton list containing just a_1. Then we allocate a second arena, a_2. Region r_2 stands for the two-element list containing a_2 and a_1. Subregioning evidence also is a *list of arenas*. It is the difference between the two lists of arenas that the regions stand for. In this example $l_2 : r_2 \sqsubseteq r_1$ is the singleton list containing a_2.

2.2 Exception Handling

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 a error condition that we want to prevent statically. Within our framework, we can phrase *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 runtime stack. For example, consider the following program.

Example 2.2. The function safeDiv divides two numbers, but throws an exception when the second number is zero.

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

We follow Zhang et al. [2016] and Brachthäuser et al. [2020a] and explicitly pass exception handlers. That is, in addition to the two parameters x and y, the function safeDiv receives an exception handler e. When y is zero we throw to this handler e. For this to be safe we need to guarantee that this handler is on the stack. But this is the very same problem we had with arenas. So we use the very same solution: When we throw to a handler of type Handler r we have to provide evidence that the current region is a subregion of the handler'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 : Handler r₁, l₁ : r₁ ⊑ ⊤) ⇒ safeDiv[r₁](5, 0, e₁) }
catch { return 0 }
```

Very much like the arena statement, the exception handler introduces a region variable r_1, a handler e_1, and subregioning evidence l_1. In the call to safeDiv, we instantiate the region variable r to r_1 and pass the exception handler e_1. The example illustrates that we can guarantee exception safety, or more generally effect safety, by the very same mechanism we use for region safety.

Meaning of Regions and Evidence. In the case of exceptions, a region is a *list of exception handlers* on the runtime stack. Evidence now corresponds to the *list of exception handlers* that an exception unwinds, again representing the difference between regions. An alternative representation that suffices in this example is to interpret evidence as the total number of exception handlers that need to be skipped over, reducing the meaning of evidence to the bare minimum.

2.3 Combining Arenas and Exceptions

Let us now look at an example where we combine arenas and exceptions.

Example 2.3. We install an exception handler and create two arenas. The inner statements are checked in region r_3.

```
try { [r₁](e₁ : Handler r₁, l₁ : r₁ ⊑ ⊤) ⇒
  arena { [r₂](a₂ : Arena r₂ String, l₂ : r₂ ⊑ r₁) ⇒
    arena { [r₃](a₃ : Arena r₃ String, l₃ : r₃ ⊑ r₂) ⇒
      alloc(a₃, "hello", 𝟘);
      alloc(a₂, "world", l₃);
      throw(e₁, l₃ ⊕ l₂)
    }
  }
} catch { return 1 }
```

To allocate into the arenas, we have to provide evidence, as before. To throw an exception to the outer handler e_1, we have to provide evidence that region r_3 is inside of r_1. We *compose* evidence variables $l_3 \oplus l_2$, to get evidence of type $r_3 \sqsubseteq r_1$.

Meaning of Regions and Evidence. In this combination, regions and evidence are again lists, with elements that are *either an arena or an exception handler*. The evidence contains exactly the arenas a_3 and a_2 we need to deallocate when we throw to handler e_1. In this example this is rather obvious. But in general, and especially in the presence of features like first-class functions, parametric polymorphism, or mutable state it is not clear that this is always the case.

2.4 Regions and Evidence

Different language features like arenas and exceptions require assigning different meaning to regions and evidence. Our conceptual framework equips us with the vocabulary to talk about evidence as the constructive difference between regions. In the remainder, we will further explore this notion of difference in two ways.

Operationally. To make the above examples precise and provide an operational intuition, in the next section we formally present the Λ_ρ calculus together with an operational semantics. In this semantics, regions and evidence are both represented as lists of markers. As we have seen in the combined example, extending a language with multiple different features requires potentially global changes to the meaning of regions and evidence.

Denotationally. To study additional language constructs and their composition, in Section 4 we present a CPS translation of Λ_ρ. Interestingly, where the small-step operational semantics represented evidence as lists, in our translation we represent regions as answer types and evidence as answer-type coercing functions, which can be understood as difference lists [Hughes 1986]. This way, each language feature can choose its own meaning of regions (by choosing a corresponding answer type) and its own meaning of evidence (by implementing the answer type coercing function). Function composition immediately gives rise to a sound composition of the different interpretations of evidence.

Terms:

Statements			
s	$::=$	$\textbf{val}\ x = s;\ s$	sequencing
	$\mid$	$\textbf{return}\ e$	returning
	$\mid$	$e[\overline{\rho}](\overline{e})$	application
Expressions			
$e,\ i$	$::=$	$x \mid f \mid l$	variables
	$\mid$	v	values
	$\mid$	$\mathbb{0}$	refl. evidence
	$\mid$	$e \oplus e$	trans. evidence
Values			
v	$::=$	$() \mid 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \ldots$	primitives
	$\mid$	$\{\ [\overline{r}](\overline{x : \tau})\ \textbf{at}\ \rho \Rightarrow s\}$	closures

Types:

Types			
τ	$::=$	$\mathsf{Int} \mid \mathsf{Bool} \mid \ldots$	primitives
	$\mid$	$\forall[\overline{r}]\ (\overline{\tau}) \to^{\rho} \tau$	functions
	$\mid$	$\rho \sqsubseteq \rho$	evidence
Regions			
ρ	$::=$	r	region variable
	$\mid$	$\top$	toplevel region

Environments:

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

Fig. 2. Syntax of our base language Λ_ρ.

3 A CALCULUS OF REGIONS – Λ_ρ

In this section, we present Λ_ρ, a calculus with regions and subregioning evidence. We then formally introduce the two extensions of the previous section: arenas and exceptions. Both of these will push markers onto the runtime stack. We call the extended language Λ_ρ [Mem, Exc]. We define a small-step operational semantics for Λ_ρ [Mem, Exc]. This semantics provides a concrete operational intuition: a region is a list of concrete *markers on the stack* and evidence is a list of *markers that represents the difference* between such lists. This allows us to establish a correspondence between type-level regions and term-level evidence, which is captured in Corollaries 3.3 and 3.4.

The paper is accompanied by a mechanized formalization of the extended language and its operational semantics in the Coq theorem prover [Bertot and Castéran 2004], including Theorems 3.1 and 3.2. Region- and exception safety follow as corollaries: whenever we use an arena or throw an exception, the corresponding marker or handler will be on the stack.

3.1 Syntax

Figure 2 defines the syntax of Λ_ρ. We use fine-grain call-by-value [Levy et al. 2003] and syntactically distinguish between statements, which can have effects, and pure expressions.

Function values (*i.e.*, $\{\ [\overline{r}](\overline{x : \tau})\ \textbf{at}\ \rho \Rightarrow s\}$) abstract over a list of type-level region parameters (*i.e.*, $\overline{r}$), and a list of term-level value 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 presentation. Our mechanized formalization includes type polymorphism, which is orthogonal to the rest of the calculus.

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

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

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 an evidence variable l, the empty evidence $\mathbb{0}$ witnessing reflexivity of subregioning, or the composition of evidence $e \oplus e$, witnessing the transitivity of subregioning.

Statement Typing.

$$\boxed{\begin{array}{c}\Gamma \mid \rho \vdash s : \tau \\ \uparrow \;\; \uparrow \;\; \uparrow \;\; \downarrow\end{array}} \qquad \frac{\Gamma \mid \rho \vdash s_0 : \tau_0 \qquad \Gamma, x_0 : \tau_0 \mid \rho \vdash s : \tau}{\Gamma \mid \rho \vdash \textbf{val } x_0 = s_0;\ s : \tau}\ [\textsc{Val}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \mid \rho \vdash \textbf{return } e : \tau}\ [\textsc{Ret}]$$

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

Expression Typing.

$$\boxed{\begin{array}{c}\Gamma \vdash e : \tau \\ \uparrow \;\; \uparrow \;\; \downarrow\end{array}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}}\ [\textsc{Lit}] \qquad \frac{\Gamma, \overline{r}, \overline{x : \tau} \mid \rho \vdash s_0 : \tau_0}{\Gamma \vdash \{\ [\overline{r}](\overline{x : \tau}) \textbf{ at } \rho \Rightarrow s_0\ \} : \forall[\overline{r}](\overline{\tau}) \to^{\rho} \tau_0}\ [\textsc{Fun}]$$

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

Fig. 3. Type system of our base language Λ_ρ.

3.2 Typing

Figure 3 defines the typing rules of 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 [1972]) computation, which is only safe in specific regions. In contrast, expressions are pure (that is, *trivial*) and can be evaluated independent of any region.

3.2.1 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 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 or explicit subtyping of function types here or elsewhere. All region subtyping exclusively occurs through the passing of subregioning evidence.

3.2.2 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. As we will see, 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.

3.2.3 Typing of Evidence. Reflexivity evidence $\mathbb{0}$ witnesses that every region is nested within itself, and transitivity 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.

Our language Λ_ρ, as presented, provides the necessary framework but does not 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. Therefore, we now introduce two extensions and then present the operational semantics.

Extended Typing Rules:

$$\frac{\Gamma,\ r,\ x\ :\ \text{Arena}\ r\ \tau',\ l\ :\ r \sqsubseteq \rho \mid r \vdash\ s\ :\ \tau}{\Gamma \mid \rho \vdash\ \textbf{arena}\ \{\ [r](x,\ l) \Rightarrow s\ \}\ :\ \tau}\ [\text{Arena}]$$

$$\frac{\Gamma \vdash\ e\ :\ \text{Arena}\ \rho'\ \tau' \quad \Gamma \vdash\ e_0\ :\ \tau' \quad \Gamma \vdash\ i\ :\ \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash\ \textbf{alloc}(e,\ e_0,\ i)\ :\ \text{Ptr}\ \rho'\ \tau'}\ [\text{Alloc}] \qquad \frac{\Gamma \vdash\ e\ :\ \text{Ptr}\ \rho'\ \tau' \quad \Gamma \vdash\ i\ :\ \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash\ \textbf{load}(e,\ i)\ :\ \tau'}\ [\text{Load}]$$

Fig. 4. Extension of $\Lambda_\rho$with arena-based memory management (Λ_ρ [Mem]).

3.3 Arenas

In this subsection, we add statements for region-based resource management. They introduce and eliminate non-trivial evidence and provide the basis for our correspondence theorem. Figure 4 introduces three additional statement forms and extends our base language to Λ_ρ [Mem].

The **arena** statement delimits a new region in which we run the statement s. It introduces three variables, a fresh region variable r, a variable $x\ :\ \text{Arena}\ r$, and evidence $l\ :\ r \sqsubseteq \rho$, witnessing that the fresh region r is a subregion of the outer region ρ. Conceptually, the **arena** statement allocates a fresh arena and pushes the pointer to this arena onto the runtime stack. It runs the statement s in this extended context to then deallocate the arena and pop the pointer. In our formalization, we refrain from modeling memory and only push and pop a fresh pointer (that is, marker).

The **alloc** statement receives an arena argument e, an initial value e_0, and an evidence argument $i\ :\ \rho \sqsubseteq \rho'$ that witnesses that the current region ρ is nested within the given region ρ'. Rule Load for **load** statements is similar. Because we do not model memory in our formalization, instead of **alloc** and **load** we have a statement **check** with the following typing rule:

$$\frac{\Gamma \vdash\ e\ :\ \text{Arena}\ \rho'\ \tau' \quad \Gamma \vdash\ i\ :\ \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash\ \textbf{check}(e,\ i)\ :\ \text{Unit}}\ [\text{Check}]$$

It asserts that the given arena is on the stack. We can implement **alloc** and **load** safely by first performing this check and then using primitives for memory operations.

3.3.1 Region Polymorphism and Subregioning Evidence. To illustrate region polymorphism and the usage of subregioning evidence let us consider a few examples.

```
def f() at T { return 5 };
arena { [r₁](a₁, l₁) ⇒ f() /* type error */ }
```

```
def f[r]() at r { return 5 };
arena { [r₁](a₁, l₁) ⇒ f[r₁]() }
```

On the left, we define a function f that has to run in the top-level region T. This example does not typecheck, since we try to call f in the fresh region r_1. If we want f to be callable in any region, we have to give it a *region-polymorphic* type, as on the right. At the call site, we have to instantiate the region parameter of f to the region r_1 in which we call it. We can constrain the region-polymorphic function to a specific region by requiring evidence as a parameter.

```
arena { [r₁](a₁, l₁) ⇒
  def f[r](l: r ⊑ r₁) at r { val u = check(a₁, l); return 5 };
  arena { [r₂](a₂, l₂) ⇒ f[r₂](l₂) }
}
```

Here we say that we can call f in any region r that is within r_1. At the call-site we instantiate r to r_2 and provide the appropriate evidence.

Extended Typing Rules:

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

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

Fig. 5. Extension of $\Lambda_\rho$with exceptions (Λ_ρ [Exc]).

Semantics of Evidence and Regions:

m	::=	`@a5f` | `@4b2` | ...	markers
w	::=	$\cdot$ | $m\ ::\ w$	evidence values
u	::=	$\cdot$ | $m\ ::\ u$	region values

Extended Syntax:

s	::=	...	
	|	$\#\textbf{arena}_m\ \{\ s\ \}$	arena marker
	|	$\#\textbf{catch}_m\ \{\ s\ \}\ \{\ s\ \}$	catch marker
v	::=	... | w	evidence value
ρ	::=	... | u	region value

Extended Typing:

$$\frac{\Gamma \mid m\ ::\ u \vdash\ s_0\ :\ \tau}{\Gamma \mid u \vdash\ \#\textbf{arena}_m\ \{\ s_0\ \}\ :\ \tau}\ [\textsc{ArenaMarker}]$$

$$\frac{\Gamma \mid m\ ::\ u \vdash\ s_0\ :\ \tau \qquad \Gamma \mid u \vdash\ s\ :\ \tau}{\Gamma \mid u \vdash\ \#\textbf{catch}_m\ \{\ s_0\ \}\ \{\ s\ \}\ :\ \tau}\ [\textsc{CatchMarker}]$$

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

Fig. 6. Run-time syntax and typing of Λ_ρ [Mem, Exc].

3.4 Exceptions

We now add statements for exceptions. We refer to the extended language as Λ_ρ [Exc]. The two new statement forms are given in Figure 5.

The **try** ... **catch** ... 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 : Handler r, and an evidence variable $l\ :\ r \sqsubseteq \rho$, witnessing that the fresh region r is a subregion of the outer region ρ. As we will see, operationally it installs a catch frame on the runtime stack, labeled with a fresh marker. The handler x contains this label in order to allow throwing to the correct exception handler.

The **throw** statement is checked similarly to **alloc**. Operationally, it throws to the given handler by unwinding the stack until it hits a catch frame with this exact marker and then executes the body of the catch clause. Again, evidence guarantees that unwinding never fails, *i.e.* the corresponding maker is always somewhere on the runtime stack.

3.5 Operational Semantics

We now define a substitution-based, small-step operational semantics for the language with both arenas and exceptions which we call Λ_ρ [Mem, Exc].

Figure 6 extends this language with run-time constructs. These do not appear in source programs but are introduced during evaluation. A region value is an ordered list of runtime markers on the runtime stack, from innermost to outermost. While in the source language regions are on the type-level, during evaluation every region will become such a region value. An evidence value is an ordered list of markers too. Rules ArenaMarker and CatchMarker type the new runtime statements. During evaluation the region we type these statements in will be a region value u. This region value is the list of markers in the evaluation context of the statement, *i.e.* the runtime

Syntax of Contexts:

$$\mathsf{K} ::= \square \mid \textbf{val}\ x = \mathsf{K};\ s \mid \#\textbf{arena}_m\ \{\ \mathsf{K}\ \} \mid \#\textbf{catch}_m\ \{\ \mathsf{K}\ \}\ \{\ s\ \}$$

Congruence:

$$\begin{array}{lcl}
\lceil . \rceil\ :\ \mathsf{K} \rightarrow u & & \\
\lceil \square \rceil & = & \cdot \\
\lceil \textbf{val}\ x = \mathsf{K};\ s \rceil & = & \lceil \mathsf{K} \rceil \\
\lceil \#\textbf{arena}_m\ \{\ \mathsf{K}\ \} \rceil & = & \lceil \mathsf{K} \rceil +\!\!+\ m \\
\lceil \#\textbf{catch}_m\ \{\ \mathsf{K}\ \}\ \{\ s\ \} \rceil & = & \lceil \mathsf{K} \rceil +\!\!+\ m
\end{array}
\qquad
\frac{s \rightarrow^{\lceil \mathsf{K} \rceil} s'}{\mathsf{K}[s] \longmapsto \mathsf{K}[s']}\ (\mathit{cong})$$

Reduction in Context:

$$\begin{array}{llcl}
(\mathit{ret}) & \textbf{val}\ x = \textbf{return}\ e;\ s & \rightarrow^u & s[x \mapsto \mathcal{V}[\![e]\!]] \\[4pt]
(\mathit{app}) & \{\ [\overline{r}](\overline{x : \tau})\ \textbf{at}\ \rho \Rightarrow s_0\ \}[\overline{u}](\overline{e}) & \rightarrow^u & s_0[\overline{r \mapsto u}][\overline{x \mapsto \mathcal{V}[\![e]\!]}] \\
 & \text{where } u\ =\ \rho[\overline{r \mapsto u}] & & \\[4pt]
(\mathit{arena}) & \textbf{arena}\ \{\ [r](x,\ l) \Rightarrow s_0\ \} & \rightarrow^u & \#\textbf{arena}_m\ \{\ s_0[r \mapsto m :: u][x \mapsto m][l \mapsto m :: \cdot\,]\ \} \\
 & \text{where } m \text{ fresh} & & \\[4pt]
(\mathit{check}) & \textbf{check}(m,\ i) & \rightarrow^u & \textbf{return}\ () \\
 & \text{where } m \in u & & \\[4pt]
(\mathit{poparena}) & \#\textbf{arena}_m\ \{\ \textbf{return}\ e\ \} & \rightarrow^u & \textbf{return}\ \mathcal{V}[\![e]\!] \\[4pt]
(\mathit{try}) & \textbf{try}\ \{\ [r](x,\ l) \Rightarrow s_0\ \}\ \textbf{catch}\ \{\ s\ \} & \rightarrow^u & \#\textbf{catch}_m\ \{\ s_0[r \mapsto m :: u][x \mapsto m][l \mapsto m :: \cdot\,]\ \}\ \{\ s\ \} \\
 & \text{where } m \text{ fresh} & & \\[4pt]
(\mathit{unwind1}) & \textbf{val}\ x = \textbf{throw}(m,\ i);\ s & \rightarrow^u & \textbf{throw}(m,\ i) \\[4pt]
(\mathit{unwind2}) & \#\textbf{arena}_{m'}\ \{\ \textbf{throw}(m,\ i)\ \} & \rightarrow^u & \textbf{throw}(m,\ w) \\
 & \text{where } \mathcal{V}[\![i]\!]\ =\ m' :: w & & \\[4pt]
(\mathit{unwind3}) & \#\textbf{catch}_{m'}\ \{\ \textbf{throw}(m,\ i)\ \}\ \{\ s\ \} & \rightarrow^u & \textbf{throw}(m,\ w) \\
 & \text{where } \mathcal{V}[\![i]\!]\ =\ m' :: w \text{ and where } m' \neq m & & \\[4pt]
(\mathit{catch}) & \#\textbf{catch}_m\ \{\ \textbf{throw}(m,\ i)\ \}\ \{\ s\ \} & \rightarrow^u & s \\
 & \text{where } \mathcal{V}[\![i]\!]\ =\ \cdot & & \\[4pt]
(\mathit{popcatch}) & \#\textbf{catch}_m\ \{\ \textbf{return}\ e\ \}\ \{\ s\ \} & \rightarrow^u & \textbf{return}\ \mathcal{V}[\![e]\!]
\end{array}$$

Evaluation of Expressions:

$$\begin{array}{lcl}
\mathcal{V}[\![\mathbb{0}]\!] & = & . \\
\mathcal{V}[\![e_1 \oplus e_2]\!] & = & \mathcal{V}[\![e_1]\!] +\!\!+\ \mathcal{V}[\![e_2]\!] \\
\mathcal{V}[\![v]\!] & = & v
\end{array}$$

Fig. 7. Operational semantics of Λ_ρ [Mem, Exc]

stack. The enclosed statement s_0 is typed in an extended runtime region $m :: u$. Rule Evidence types evidence values and connects run-time evidence, type-level regions, and run-time regions. At runtime, the evidence value w, the runtime region u_0 and the runtime region u_1 will all be lists of markers. The evidence w is precisely the difference between the runtime regions u_0 and u_1. Our proof of preservation ensures that this invariant always holds throughout reduction.

3.5.1 Reduction Semantics. Figure 7 defines a small-step operational semantics for Λ_ρ [Mem, Exc]. It is all about that stack: the evaluation context K directly models the runtime stack with normal stack frames, arena markers, and catch clauses. The crucial rule is the one for congruence *(cong)*. It defines the reduction relation of statements $\longmapsto$ in terms of a reduction relation $\rightarrow^{u}$, where u is a run-time region extracted from the actual context as $\lceil \mathsf{K} \rceil$. Indexing the reduction by the current runtime region allows us to establish the correspondence between regions as they appear on the type level and the concrete region as the list of markers on the runtime stack at run time.

Since evaluation of expressions does not affect the evaluation context, we present its reduction as a big-step reduction rule $\mathcal{V}[\![\,\cdot\,]\!]$. The reflexivity evidence is the empty list of markers and transitivity of evidence appends the two lists of markers. If we had (trivial) primitives like addition we would define their reduction in this rule, too. Rule *(ret)* is fairly standard. In the rule for function application *(app)*, we check that the annotated region ρ matches exactly (after substitution) the actual run-time region. A non matching region results in a stuck term.

In rule *(arena)*, we create a fresh marker m and run the statement s in a context with this new marker added. We substitute the extended runtime region $(m :: u)$ for the region variable r. We substitute the fresh marker m for the arena variable x. The evidence variable l witnesses the nesting of r in ρ by describing the difference between the two runtime regions as a singleton list of the fresh marker m. Rule *(check)* asserts that the marker m is an element of the current runtime region u. It returns the unit value when this check succeeds and gets stuck otherwise.

Rule *(try)* does the same as rule *arena*, but pushes a catch frame onto the stack instead of an arena frame. When an exception is thrown we unwind the stack frame by frame until we find the matching catch frame. We pop elements off the evidence i in lock step with popping arena- and catch frames off the stack. We assert that we 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.

We mechanized the formalization of Λ_ρ [Mem, Exc] in the Coq theorem prover and showed the usual theorems of progress and preservation.

Theorem 3.1 (Progress).
If $\emptyset \mid \cdot \vdash s : \tau$*, then either* $s \longmapsto s'$ *or* s *is of the form* **return** v *for some value* v*.*

Theorem 3.2 (Preservation).
If $\emptyset \mid \cdot \vdash s : \tau$ *and* $s \longmapsto s'$ *then* $\emptyset \mid \cdot \vdash s' : \tau$*.*

3.6 Region- and Evidence Correspondence

The goal of this section was to formally establish the connection between type-level regions and term-level evidence in the presence of region-based resource management (arenas) and control effects (exceptions) during evaluation. We've set everything up so the following two simple corollaries hold by construction:

Corollary 3.3 (Region Correspondence).
If $\emptyset \mid \cdot \vdash \mathsf{K}[s] : \tau$*, then* $\emptyset \mid \lceil K \rceil \vdash s : \tau'$ *for some type* τ'*.*

Corollary 3.4 (Evidence Correspondence).
If an evidence values w *has type* $u_0 \sqsubseteq u_1$ *for some runtime regions* u_0 *and* u_1 *then* $u_0 = w +\!\!+\, u_1$*.*

Type-level region variables stand for exactly the lists of markers that the current runtime context contains. Evidence values are exactly the difference between two such lists. These corollaries are inspired by the similarly named theorem of Xie et al. [2020].

Together, these corollaries make runtime regions and runtime evidence on the one hand and marker frames on the stack on the other hand redundant: We could erase regions and evidence as they do not have any significance at runtime. In the next section we are going to do the opposite: Erase marker frames and solely rely on evidence terms to have the correct content at runtime.

Translation of Types:

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

Translation of Statements:

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

Translation of Expressions:

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

Auxiliary Definitions:

$$\textsc{Cps}\ R\ A \quad = \quad (A \rightarrow R) \rightarrow R$$

Fig. 8. Translation from $\Lambda_{\rho}$to System F.

4 COMBINING REGIONS AND EFFECTS VIA CONTINUATION-PASSING STYLE

We now give a denotational semantics to Λ_{ρ} [Mem, Exc] as a translation to System F in CPS. Again, it is all about that stack: evaluation contexts K now become continuations [Danvy 2004]. In our translation, regions correspond to *answer types* and evidence terms are translated to *answer-type coercions*, generalizing the interpretation of evidence by Fluet and Morrisett [2004]. By translation into CPS, the semantics is easily extensible, allowing us to present multiple different extensions that naturally can be composed without changing the denotations of the others.

Our translation to System F can also serve as a compilation technique for languages with effects and resources 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. [2020], which has been shown to enable compile-time optimizations for significant performance improvements.

We implemented Λ_{ρ} and all of the extensions and examples of this section in the dependently typed language Idris 2 [Brady 2020] as a shallow embedding. This serves to establish two things: Firstly, the translations take well-typed terms to well-typed terms in System F and soundness directly follows from the soundness of System F. Secondly, running the examples yields the expected results indicating that all language constructs plausibly do what they should under this semantics.

4.1 Translation of the Base Language Λ_{ρ}

To give semantics to Λ_{ρ} and to its extensions, we translate them to System F in one particular variant of CPS, called *iterated CPS* [Danvy and Filinski 1990; Schuster and Brachthäuser 2018]. That is, we use a more structured form of continuations (or stacks). Every stack segment, delimited by a marker, 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. Figure 8 defines the translation of Λ_ρ to System F. 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 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*. 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 3.5. In general, statements with return type τ that have to be run in a region ρ are translated to terms of type $\textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ \mathcal{T}[\![\tau]\!]$. This can for instance be seen in the translation of function types.

The semantics of regions and evidence are as follows.

(1) We translate regions to answer types. Each of the extensions is free to choose a different answer type. Region abstractions are translated to type abstractions and region polymorphic functions have a polymorphic answer type. To make the connection to our overview in Figure 1, throughout the current section, we will highlight the choice of answer types as τ.

(2) We translate evidence expressions to functions that lift a computation to run in a different region. The concrete implementation of evidence is again up to the specific extension. Generally, the reflexivity evidence is translated to the polymorphic identity function, and transitivity of evidence amounts to function composition. Again, to make the connection to our overview, we highlight the denotational choice of evidence as $\Lambda a \Rightarrow \lambda m \Rightarrow \ldots$.

These two aspects represent the key novelty of our translation. To emphasize, the translated type of evidence $\forall a.\ \textsc{Cps}\ \mathcal{T}[\![\ \rho'\ ]\!]\ a \to \textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ a$ tells us that evidence transforms a computation to have a different answer type! Comparing this type to our interpretation of evidence in the previous section, it not merely tells us the difference between two regions, it actually allows us to *move* between regions on the stack. This interpretation is also in accordance with the visualization of Figure 1: evidence tells us how to do a control transfer from one region to another.

We translate well-typed programs in Λ_ρ and all of its extensions presented in this section to well-typed programs in System F. The accompanying Idris code is a proof of this.

Theorem 4.1 (Well-typedness of Translated Terms).
If $\Gamma \mid \rho \vdash\ s\ :\ \tau$, *then* $\mathcal{T}[\![\ \Gamma\ ]\!] \vdash\ \mathcal{S}[\![s]\!]_\rho\ :\ (\mathcal{T}[\![\ \tau\ ]\!] \to \mathcal{T}[\![\ \rho\ ]\!]) \to \mathcal{T}[\![\ \rho\ ]\!]$

In the remainder of this section, we extend our base language Λ_ρ and define translations for different language features, such as exceptions, finalizers, effect handlers, mutable state, or dynamic-wind. All translations are based on the idea of regions as answer types and evidence terms as answer-type coercions. The translation of the base language stays the same.

4.2 Arenas

In Figure 4, we have seen the typing rules for Λ_ρ [Mem]. Now we give translation of this feature to CPS. In general, the same approach works also for resources other than memory, such as file handles. In Section 4.4 we present a more general finalization construct.

Translation. Figure 9 defines the translation of Λ_ρ [Mem] to CPS. We do not need any runtime checks to prevent pointers from being used outside of their region, justified by Corollaries 3.3 and 3.4. The **arena** statement creates a fresh arena. When control leaves the enclosed block, memory

Extended Translation Rules:

$$\mathcal{T}[\![\text{Arena } \rho\ \tau]\!] = \texttt{PrimArena}$$
$$\mathcal{T}[\![\text{Ptr } \rho\ \tau]\!] = \texttt{PrimPtr}$$

$$\mathcal{S}[\![\textbf{arena } \{ [r](x,\ l) \Rightarrow s_0 \}]\!]_\rho =$$
$$\qquad \textsc{RunArena}\quad (\lambda h \Rightarrow (\Lambda r \Rightarrow \lambda x \Rightarrow \lambda l \Rightarrow \mathcal{S}[\![s_0]\!]_r)\quad (\mathcal{T}[\![\rho]\!])\quad h\quad (\textsc{LiftArena}\ h)\ )$$

$$\mathcal{S}[\![\textbf{alloc}(e,\ e_0,\ i)]\!]_\rho = \lambda k \Rightarrow k\ (\texttt{allocPrimPtr}\ \mathcal{E}[\![e]\!]\ \mathcal{E}[\![e_0]\!])$$
$$\mathcal{S}[\![\textbf{load}(e,\ i)]\!]_\rho = \lambda k \Rightarrow k\ (\texttt{loadPrimPtr}\ \mathcal{E}[\![e]\!])$$

Auxiliary Definitions:

$$\textsc{RunArena} : (\texttt{PrimArena} \to \textsc{Cps}\ R\ A) \to \textsc{Cps}\ R\ A$$
$$\textsc{RunArena} = \lambda m \Rightarrow \lambda k \Rightarrow$$
$$\qquad \textbf{let}\ h = \texttt{allocPrimArena}\ ();\ m\ h\ (\lambda x \Rightarrow \texttt{freePrimArena}\ h;\ k\ x)$$

$$\textsc{LiftArena}\ h : \forall a.\ \textsc{Cps}\ R\ a \to \textsc{Cps}\ R\ a$$
$$\textsc{LiftArena}\ h = \Lambda a \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow \texttt{freePrimArena}\ h;\ m\ k$$

Fig. 9. Translation of $\Lambda_\rho$with arena-based memory management (Λ_ρ [Mem]).

in the arena is destroyed. In its translation we use the auxiliary meta function RunArena. It binds the current continuation k and allocates a primitive arena h. We run the given computation m with h and a continuation where we push a frame that frees the arena onto the current continuation k. In the body of the arena statement we abstract over a region r, an arena x, and evidence l. We apply this function to three arguments: The outer region ρ, the primitive arena h, and the evidence LiftArena h. Importantly, this evidence will free the arena. This corresponds to allocating a specialized frame on the stack for finalization. The **alloc** statement allocates a value into an arena. We require evidence that the arena is still live, *i.e.* on the runtime stack, but don't actually use it. Similarly, when we use the **load** statement to load a value from a pointer, we require evidence that the corresponding arena is still live, which it always is. Evidence terms are translated to functions (Section 4) whereas in Section 3.5 evidence was a list of markers. However, under our translation, evidence still contains a list of markers. It is just hidden in the closure environment of the evidence. Evidence composition concatenates these lists.

Example 4.2. 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 in the right. It has type Cps Void Int.

```
arena {
  [r₁](a₁: Arena r₁ A, l₁: r₁ ⊑ T) ⇒
    val ptr = alloc(a₁, aValue, 𝟘);
    return 0
}
```

$$\begin{array}{l} \lambda k \Rightarrow \\ \quad \textbf{let}\ h = \texttt{allocPrimArena}\ (); \\ \quad (\Lambda r_1 \Rightarrow \lambda a_1 \Rightarrow \lambda l_1 \Rightarrow \lambda k_1 \Rightarrow \\ \quad\quad \textbf{let}\ \mathit{ptr} = \texttt{allocPrimPtr}\ a_1\ \mathit{aValue}; \\ \quad\quad k_1\ 0)\ \text{Void}\ h \\ \quad\quad\quad (\Lambda a \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow \texttt{freePrimArena}\ h;\ m\ k) \\ \quad\quad\quad (\lambda x \Rightarrow \lambda k \Rightarrow \texttt{freePrimArena}\ h;\ k\ x) \end{array}$$

This term can be simplified to the following

$$\lambda k \Rightarrow \textbf{let}\ h = \texttt{allocPrimArena}\ ();\ \textbf{let}\ \mathit{ptr} = \texttt{allocPrimPtr}\ h\ \mathit{aValue};\ \texttt{freePrimArena}\ h;\ k\ 0$$

So far, we have not used any evidence yet. All control flow is local and the only way to leave a region is to return from it. In the next section we will add exceptions as an example of a control operator with non-local control transfer.

Extended Translation Rules:

$$\mathcal{T}[\![\ \text{Handler}\ \rho]\!] \quad = \quad \textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \text{Void}$$

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

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

Auxiliary Definitions:

$\textsc{RunCps}$	:	$\textsc{Cps}\ (\textsc{Cps}\ R\ A)\ A \longrightarrow \textsc{Cps}\ R\ A$
$\textsc{RunCps}$	=	$\lambda m \Rightarrow m\ (\lambda x \Rightarrow \lambda k \Rightarrow k\ x)$
$\textsc{LiftCps}$	:	$\forall a.\ \textsc{Cps}\ R\ a \longrightarrow \textsc{Cps}\ (\textsc{Cps}\ R\ R')\ a$
$\textsc{LiftCps}$	=	$\Lambda a \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow \lambda j \Rightarrow m\ (\lambda x \Rightarrow k\ x\ j)$

Fig. 10. Translation of $\Lambda_\rho$with exceptions (Λ_ρ [Exc]).

4.3 Exceptions

Figure 5 presented the extension of Λ_ρ with exceptions (Λ_ρ [Exc]). In Section 3.5 we have seen an operational semantics for this language. Now we present translation for this language to System F. Whereas in the operational semantics 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.

Translation. Figure 10 defines the semantics of exceptions as a translation to System F. It generalizes the translation to double-barreled CPS from [Kennedy 2007] to iterated CPS. In the translation of the **try** ... **catch** ... statement, we use RunCps. It runs the given computation m with an additional continuation which is initially empty. We instantiate the answer type r of the translated body s_0 to be the type $\textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ \mathcal{T}[\![\tau]\!]$. A Handler ρ is a CPS expression that aborts the current continuation. 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. In the translation of a **throw**$(e,\ i)$ statement, to abort the current continuation, we now use the evidence to lift the handler into the correct region. This way, the handler can run in the current region and is compatible to the current answer type.

Example 4.3. Let us consider the motivating example with exceptions from Section 2.2. Again, the example on the left is translated to the resulting System F term of type Cps Void Int on the right.

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

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

The resulting System F term can be further simplified to:

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

Extended Typing Rules:

$$\frac{\Gamma,\ r,\ l :\ r \sqsubseteq \rho \mid r \vdash s_0\ :\ \tau \qquad \Gamma \mid \rho \vdash s\ :\ ()}{\Gamma \mid \rho \vdash\ \textbf{try}\ \{\ [r](l) \Rightarrow s_0\ \}\ \textbf{unwind}\ \{\ s\ \}\ :\ \tau}\ [\textsc{Unwind}]$$

Extended Translation Rules:

$$\mathcal{S}[\![\ \textbf{try}\ \{\ [r](l) \Rightarrow s_0\ \}\ \textbf{unwind}\ \{\ s\ \}\ ]\!]_\rho\ =$$
$$(\Lambda r \Rightarrow \lambda l \Rightarrow \mathcal{S}[\![\ s_0\ ]\!]_r)\quad \mathcal{T}[\![\ \rho\ ]\!]\quad (\textsc{LiftFin}\ \mathcal{S}[\![\ s\ ]\!]_\rho)$$

Auxiliary Definitions:

$$\begin{array}{lll} \textsc{LiftFin}\ f & : & \forall a.\ \textsc{Cps}\ R\ a \rightarrow \textsc{Cps}\ R\ a \\ \textsc{LiftFin}\ f & = & \Lambda a \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow f\ (\lambda u \Rightarrow m\ k) \end{array}$$

Fig. 11. Extension of $\Lambda_\rho$with finalizers (Λ_ρ [Fin]).

We instantiate the answer type r of `safeDiv` with r_1 which we then instantiate to the type Cps Void Int. Its overall return type is then Cps (Cps Void Int) Int. It receives two continuations. The exception handler e_1 discards the first continuation and returns 0 to the second.

Our translation to CPS is compositional and as such interacts nicely with the evidence terms we have defined for arenas in Section 4.2: We free an arena exactly when an exception is thrown across it. We now generalize this idea and run a user-defined cleanup action whenever we leave a region.

4.4 Finalizers

In Figure 11, we again extend Λ_ρ, this time with finalizers (Λ_ρ [Fin]). The **try** ... **unwind** ... statement runs the action s whenever an exception would cause control to leave the corresponding **try**. The cleanup action will not be called upon normal return.

Translation. In the translation, the evidence LiftFin f will run the cleanup statement f, ignore its result (bound to u), and continue to run the rest of the evidence m. This is a generalization of the translation of the **arena** statement: Instead of freeing the arena, we run an arbitrary user-defined cleanup action. The evidence contains this cleanup action. When we compose evidence it will conceptually contain a list of cleanup actions that are all run sequentially.

Example 4.4. The following example illustrates what happens in our semantics when a finalizer itself throws an exception.

```
try { [r₁](e₁ : Handler r₁, l₁ : r₁ ⊑ ⊤) ⇒
  try { [r₂](l₂ : r₂ ⊑ r₁) ⇒
    throw(e₁, l₂)
  } unwind {
    throw(e₁, 𝟘)
  }
} catch { return 0 }
```

To throw from region r_2 to region r_1, we have to provide evidence that $r_2 \sqsubseteq r_1$. It contains the finalization function which will be called. The unwind statement runs in region r_1, to throw from it we have to provide evidence that $r_1 \sqsubseteq r_1$. Because the finalizer is part of the evidence, throwing from it will abort finalization and start to run the evidence given to the `throw` statement.

In the following subsection, we look at a control operator that exposes the delimited continuation of the current computation to the programmer.

Extended Typing Rules:

$$\frac{\Gamma,\ r,\ x\ :\ \text{Prompt}\ r\ \rho\ \tau,\ l\ :\ r \sqsubseteq \rho \mid r \vdash\ s\ :\ \tau}{\Gamma \mid \rho \vdash\ \textbf{reset}\ \{\ [r](x,\ l) \Rightarrow s\ \}\ :\ \tau}\ [\textsc{Reset}]$$

$$\frac{\Gamma \vdash\ e\ :\ \text{Prompt}\ \rho'\ \rho_0\ \tau_0 \quad \Gamma \vdash\ i\ :\ \rho \sqsubseteq \rho' \quad \Gamma,\ k\ :\ (\tau) \rightarrow^{\rho_0} \tau_0 \mid \rho_0 \vdash\ s_0\ :\ \tau_0}{\Gamma \mid \rho \vdash\ \textbf{shiftTo}(e,\ i)\ \{\ (k) \Rightarrow s_0\ \}\ :\ \tau}\ [\textsc{ShiftTo}]$$

Extended Translation Rules:

$$\mathcal{T}[\![\ \text{Prompt}\ \rho\ \rho_0\ \tau_0\ ]\!] \quad = \quad \forall a.\ \textsc{Cps}\ (\textsc{Cps}\ \mathcal{T}[\![\rho_0]\!]\ \mathcal{T}[\![\tau_0]\!])\ a \rightarrow \textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ a$$

$$\mathcal{S}[\![\ \textbf{reset}\ \{\ [r](x,\ l) \Rightarrow s\ \}\ ]\!]_\rho \quad =$$

$$\textsc{RunCps}\ ((\Lambda r \Rightarrow \lambda x \Rightarrow \lambda l \Rightarrow \mathcal{S}[\![\ s\ ]\!]_r)\ \ (\textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \mathcal{T}[\![\ \tau\ ]\!])\ \ (\Lambda a \Rightarrow \lambda m \Rightarrow m)\ \ (\textsc{LiftCps})\ )$$

$$\mathcal{S}[\![\ \textbf{shiftTo}(e,\ i)\ \{\ (k) \Rightarrow s_0\ \}\ ]\!]_\rho \quad = \quad \mathcal{E}[\![i]\!]\ \ (\mathcal{T}[\![\tau]\!])\ \ (\mathcal{E}[\![e]\!]\ \ (\mathcal{T}[\![\tau]\!])\ \ (\lambda k \Rightarrow \mathcal{S}[\![\ s_0\ ]\!]_{\rho_0}))$$

Fig. 12. Extension of $\Lambda_\rho$with a control operator (Λ_ρ [Shf]).

4.5 Control Operators

Whereas exceptions allow us to jump out of a region, more general control operators allow us to jump back into it. There are many flavors of control operators that capture the continuation and give programmers the choice to resume computation, perhaps multiple times, or discard the continuation and abort [Danvy and Filinski 1990; Dybvig et al. 2007; Felleisen 1988; Sitaram and Felleisen 1990]. They are useful for structuring programs with complex control flow and can express a large number of useful idioms [Haynes 1987; Hieb and Dybvig 1990; Leijen 2017].

Figure 12 extends Λ_ρ with an operator for delimited control (Λ_ρ [Shf]). We install a delimiter with **reset**. The body s runs in a fresh region r. The body is also given access to a prompt x of type Prompt $r\ \rho\ \tau$. It witnesses that when we are in region r and we can shift to the outer region ρ with an answer type τ. Finally, the body has access to evidence l that the fresh region r is inside of the outer region ρ. We capture the current continuation with **shiftTo**, which has three arguments: the prompt e that we want to capture the continuation to, evidence i that (conceptually) this prompt is currently on the stack, and a body s that can use the current continuation k. While the overall statement runs in region ρ, the body runs in region ρ_0 and has to answer with a type τ_0. Also, the continuation k has to run in the very same region ρ_0 and will return an answer of type τ_0.

Translation. The translation uses iterated CPS in a way very similar to how we translated exceptions in Section 4.3. To translate the delimiter **reset**, we again don't segment the stack with markers, but have multiple stacks (*i.e.* continuations). The evidence explains how to capture the correct number of continuations. Differently to exceptions, however, is that instead of running a fixed handler we now run the given body s_0 in the translation of **shiftTo**. We do not discard the continuation k, it can occur free in the translated body s_0.

Example 4.5. The following classical example by Danvy and Filinski [1990] uses delimited control to compose the current continuation with itself $1\ +\ \textbf{reset}(10\ +\ \textbf{shift}\ (\lambda k \Rightarrow k\ (k\ 100)))$. Calling the continuation twice, duplicates the frame $10\ +\ \Box$ and thus running the example evaluates to $1\ +\ (10\ +\ (10\ +\ 100))\ =\ 121$. Translated to Λ_ρ [Shf] it looks like this:

```
1 + reset { [r₁](p₁, l₁) ⇒ 10 + shiftTo(p₁, 𝟘) { (k) ⇒ k(k(100)) } }
```

Let us assume this statement runs in a region r_0. Then the prompt p_1 has type Prompt $r_1\ r_0$ Int. We can use it in region r_1 (and any subregion) to capture the current continuation and jump back into

Extended Typing Rules:

$$\frac{\Gamma,\ r,\ f :\ \mathsf{Cap}\ r\ \tau_1\ \tau_2,\ l :\ r \sqsubseteq \rho \mid r \vdash\ s_0\ :\ \tau \qquad \Gamma,\ x :\ \tau_1,\ k :\ \mathsf{Cap}\ \rho\ \tau_2\ \tau \mid \rho \vdash\ s\ :\ \tau}{\Gamma \mid \rho \vdash\ \textbf{try}\ \{\ [r](f,\ l) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}\ :\ \tau}\ [\textsc{Try}]
\qquad
\frac{\Gamma \vdash\ e_0\ :\ \mathsf{Cap}\ \rho'\ \tau_1\ \tau_2 \quad \Gamma \vdash\ e\ :\ \tau_1 \quad \Gamma \vdash\ i\ :\ \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash\ \textbf{do}(e_0,\ e,\ i)\ :\ \tau_2}\ [\textsc{Do}]$$

Extended Translation Rules:

$$\mathcal{T}[\![\ \mathsf{Cap}\ \rho\ \tau_1\ \tau_2]\!] \quad = \quad \mathcal{T}[\![\tau_1]\!] \to \textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ \mathcal{T}[\![\tau_2]\!]$$

$$\mathcal{S}[\![\ \textbf{do}(e_0,\ e,\ i)\ ]\!]_\rho \quad = \quad \mathcal{E}[\![i]\!]\ \ (\mathcal{T}[\![\tau_2]\!])\ \ (\mathcal{E}[\![e_0]\!]\ \mathcal{E}[\![e]\!])$$

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

Fig. 13. Extension of $\Lambda_\rho$with effect handlers (Λ_ρ [Eff]).

region r_0. The type of the result of **reset**, *i.e.* the answer type, is `Int`. In the body of **shiftTo** we can safely use everything we can use in region r_0. The continuation k has type $(\mathsf{Int}) \to^{\rho_0} \mathsf{Int}$, signaling that it runs in region ρ_0, the one outside of the **reset**. Both calls to the continuation ρ_0 happen in exactly that region.

The control operator `shiftTo` that we introduced here, is more limited than other control operators. We statically enforce the restriction of *scoped resumptions* [Xie et al. 2020], that is, the continuation will always be called in exactly the same region that it was based in. For the same reason, even though we use the word "Prompt", this is very different from multi-prompt delimited control [Sitaram and Felleisen 1990]. In the next subsection we will look at a different way to access delimited continuations: effect handlers.

4.6 Effect Handlers

Among the different approaches to effect handlers, the one that fits particularly nicely with our framework are effect handlers in *capability-passing style* [Brachthäuser and Schuster 2017; Brachthäuser et al. 2020a]. In this style, an effect handler delimits the current continuation and provides a capability that will capture the current continuation up to the corresponding handler. Effect safety means that this capability shall only be used within the dynamic extent of the handler.

Figure 13 extends Λ_ρ with statements and typing rules for effect handlers, resulting in the language Λ_ρ [Eff]. The **try** ... **with** ... statement for effect handlers is very similar to the one for exception handlers. The delimited statement s_0 is typed in a fresh region r. It gets access to a capability $f\ :\ \mathsf{Cap}\ r\ \tau_1\ \tau_2$. This capability can be used in region r and in regions nested in it, and can be applied to an argument of type τ_1 to get a result of type τ_2.

The statement s in the handler clause gets access to a parameter x and a continuation k. We model the continuation also as a capability, as we can only use it in region ρ and any region nested in ρ. This restriction is important to guarantee effect safety. The continuation might itself use effect operations and we want to guarantee that the corresponding delimiters are on the stack when we call the continuation. When we use a capability with $\textbf{do}(e_0,\ e,\ i)$, we supply an argument e and evidence i that the current region ρ is nested in the region of the capability ρ'.

Translation. Figure 13 defines the semantics of effect handlers as a translation to iterated CPS. Capabilities are (effectful) functions. The translated **try** ... **with** ... statement installs a delimiter. Rather than always discarding the continuation, as was the case in exception handlers, the translated handler clause s can make use of it. When we perform an effect operation, we apply the translated capability to the translated argument and use the translated evidence to lift the resulting computation to run in the correct region.

Example 4.6. The following example uses effect handlers to fork the current computation.

```
def handleForkList[r₀, t](
  prog : [r](Cap r () Bool, r ⊑ r₀) →r t
) at r₀ {
  try { [r₁](fork, l₁) ⇒
    val result = prog[r₁](fork, l₁);
    return (singletonList(result))
  } with { ((), resume) ⇒
    val xs = do(resume, true, 𝟘);
    val ys = do(resume, false, 𝟘);
    append(xs, ys)
}
```

The function `handleForkList` provides the capability `fork` to the given program `prog`. When we perform a fork, we capture the current continuation and resume twice: once with `true` and once with `false` and append the resulting lists.

4.7 Local State

Another interesting use-case for region tracking is local mutable state [Launchbury and Peyton Jones 1994]. The idea is that we can use mutable references locally but encapsulate this mutation so that the overall function is pure [Timany et al. 2017]. For this to be safe, again, it is key that mutable references are not used outside of their region either directly or through an escaping function that has closed over them. The typing judgements of Λ_ρ [State] (Figure 14), the extension of Λ_ρ with local mutable references, ensure this. It is possible to implement references as raw pointers into the global heap and perform updates in-place for increased performance. However, naïvly combining effect handlers and this implementation of local mutable references results in undesired consequences.

Example 4.7. The interaction of effect handlers and local mutable references is illustrated in the following example.

```
handleForkList[T, Int]({ [r₁](fork, l₁) ⇒
  new(8) { [r₂](ref, l₂) ⇒
    val b = do(fork, (), l₂);
    if (b) { val x = get(ref, 𝟘); set(ref, x + 1, 𝟘) }
    else   { val x = get(ref, 𝟘); set(ref, x + 2, 𝟘) };
    get(ref, 𝟘)
  }})
```

We handle this program with the handler function for `fork` that we've seen in Subsection 4.6. If we use global mutable references, for example a heap allocated reference cell, we would get the list `[9, 11]` as the result of running this program, which is wrong. This program should return the list `[9, 10]`. We know that the fork is across the local reference `ref`, because we have to pass evidence `l₂` when we perform the fork.

When we fork the computation, it is important that modifications to references are only performed locally in the current branch of the computation [Kiselyov et al. 2006; Pauwels et al. 2019]. Moreover,

Extended Typing Rules:

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma,\, r,\, x : \mathbf{Ref}\; r\; \tau_0,\; l : r \sqsubseteq \rho \mid r \vdash s : \tau}{\Gamma \mid \rho \vdash \mathbf{new}(e_0)\; \{\, [r](x,\, l) \Rightarrow s \,\} : \tau}\;[\textsc{New}]$$

$$\frac{\Gamma \vdash e_0 : \mathbf{Ref}\; \rho'\; \tau \quad \Gamma \vdash i : \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash \mathbf{get}(e_0,\, i) : \tau}\;[\textsc{Get}] \qquad \frac{\Gamma \vdash e_0 : \mathbf{Ref}\; \rho'\; \tau \quad \Gamma \vdash e : \tau \quad \Gamma \vdash i : \rho \sqsubseteq \rho'}{\Gamma \mid \rho \vdash \mathbf{set}(e_0,\, e,\, i) : ()}\;[\textsc{Set}]$$

Extended Translation Rules:

$$\mathcal{T}[\![\, \mathbf{Ref}\; \rho\; \tau \,]\!] = (\text{Unit} \to \textsc{Cps}\; \mathcal{T}[\![\rho]\!]\; \mathcal{T}[\![\tau]\!]) \times (\mathcal{T}[\![\tau]\!] \to \textsc{Cps}\; \mathcal{T}[\![\rho]\!]\; \text{Unit})$$

$$\mathcal{S}[\![\, \mathbf{new}(e_0)\; \{\, [r](x,\, l) \Rightarrow s \,\} \,]\!]_\rho =$$
$$\textsc{RunState}\;\; \mathcal{E}[\![e_0]\!]\;\; ((\Lambda r \Rightarrow \lambda x \Rightarrow \lambda l \Rightarrow \mathcal{S}[\![\, s \,]\!]_r)\;\; (\textsc{Reader}\; \mathcal{T}[\![\tau_0]\!]\; \mathcal{T}[\![\rho]\!])\;\; (\textsc{Get},\, \textsc{Set})\;\; (\textsc{LiftState})\,)$$

$$\mathcal{S}[\![\, \mathbf{get}(e_0,\, i) \,]\!]_\rho = \mathcal{E}[\![i]\!]\; \mathcal{T}[\![\tau]\!]\; (\mathit{fst}\; \mathcal{E}[\![e_0]\!]\; ())$$
$$\mathcal{S}[\![\, \mathbf{set}(e_0,\, e,\, i) \,]\!]_\rho = \mathcal{E}[\![i]\!]\; \mathcal{T}[\![\text{Unit}]\!]\; (\mathit{snd}\; \mathcal{E}[\![e_0]\!]\; \mathcal{E}[\![e]\!])$$

Auxiliary Definitions:

$$\textsc{Reader}\; S\; A = S \to A$$

$$\textsc{RunState} : S \to \textsc{Cps}\; (\textsc{Reader}\; S\; R)\; A \to \textsc{Cps}\; R\; A$$
$$\textsc{RunState} = \lambda z \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow m\; (\lambda x \Rightarrow \lambda s \Rightarrow k\; x)\; z$$

$$\textsc{LiftState} : \forall a.\; (\textsc{Cps}\; R\; a) \to (\textsc{Cps}\; (\textsc{Reader}\; S\; R)\; a)$$
$$\textsc{LiftState} = \Lambda a \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow \lambda s \Rightarrow m\; (\lambda x \Rightarrow k\; x\; s)$$

$$\textsc{Get} : \text{Unit} \to \textsc{Cps}\; (\textsc{Reader}\; S\; R)\; S$$
$$\textsc{Get} = \lambda u \Rightarrow \lambda k \Rightarrow \lambda s \Rightarrow k\; s\; s$$

$$\textsc{Set} : S \to \textsc{Cps}\; (\textsc{Reader}\; S\; R)\; \text{Unit}$$
$$\textsc{Set} = \lambda s \Rightarrow \lambda k \Rightarrow \lambda d \Rightarrow k\; ()\; s$$

Fig. 14. Extension of $\Lambda_\rho$with local state (Λ_ρ [State]).

modifications to other references, created outside of the handler for `fork`, should still be visible to both branches of the computation. Our framework suggests two solutions to this problem.

Translation. Figure 14 presents the semantics of local mutable references as a translation from Λ_ρ [State] to System F, that exhibits the correct backtracking behavior. When we introduce a new mutable reference with **new**, we run the inner computation at type Cps (Reader E R) A. A reference is a pair of a getter and a setter. The evidence term will push the current state onto the continuation. Again, it is “all about that stack”. Conceptually, we put local state onto the stack and we do so by an appropriate choice of answer type. This translation dictates the correct semantics, but it would be wasteful in practice, because we use the evidence in all **get** and **set** statements which makes access to references linear in the number of nested reference handlers, and moreover, when combined with other extensions, runs all computations contained in the evidence. However, this translation is useful for local optimization of functions that use mutable references when compiling with continuations [Cong et al. 2019]. Since we translate to pure System F, we reduce the problem of optimizing mutable references to the problem of optimizing pure functions. We effectively transform local mutable references to use registers.

Example 4.8. As an example for such optimizations consider the statement:

```
val x = get(ref, 𝟘); set(ref, x + 1, 𝟘)
```

Using the translation of Λ_ρ [State] to System F the program translates to

$$\begin{array}{l} \lambda j \Rightarrow ((\Lambda a \Rightarrow \lambda m \Rightarrow m)\ \mathsf{Int}\ ((\lambda u \Rightarrow \lambda k \Rightarrow \lambda s \Rightarrow k\ s\ s)\ ()) \\ \quad (\lambda x \Rightarrow (\Lambda a \Rightarrow \lambda m \Rightarrow m) \\ \qquad \mathsf{Unit}\ \ ((\lambda s \Rightarrow \lambda k \Rightarrow \lambda d \Rightarrow k\ ()\ s)\ (x\ +\ 1)))\ \ j) \end{array}$$

and can statically be reduced to $\lambda j \Rightarrow \lambda s \Rightarrow j\ ()\ (s\ +\ 1)$.

Under our translation, optimization of effectful programs is just inlining and β-reduction, which is a well-studied topic in compilers for functional languages. This optimization interacts well with control effects by iterated CPS. In fact, we can fully reduce Example 4.7 at compile time to its result just by inlining and β-reduction. This is similar to what Schuster et al. [2020] propose, but our source language and translation are more general.

Backtracking State via Backup/Restore. Another solution to the problem of correctly backtracking mutable reference in the presence of control, is to store references on the runtime stack, backup-up the current state during unwinding, and restore that state upon resumption [Brachthäuser et al. 2018, 2020b; Kiselyov et al. 2006]. In our translation, evidence has computational content. This way, as an alternative implementation strategy, we can use global mutable references without any special runtime support and still obtain the correct backtracking behavior. The idea is to push the code that performs the backup and restore into the evidence. When we capture a continuation across a mutable reference, the continuation closes over the state, which it restores on resumption. Please consult the accompanying code for more details. This has advantages from a performance perspective: code that does not capture the continuation only performs reads and writes on raw references. At the same time, code that does capture the continuation has the correct backtracking behavior. In the next section we will generalize this idea and present dynamic wind.

4.8 Dynamic Wind

Dynamic wind [Friedman and Haynes 1985] allows user definable actions to be performed every time control effects are used to enter and leave a region. We have already seen the first half of dynamic wind in Λ_ρ [Fin], where we could install a cleanup action that is run when we unwind the stack across it. Naturally, in our framework finalizers also interact well with effect handlers and can be combined to obtain the language Λ_ρ [Eff, Fin]. However, in face of resumable exceptions, such as provided by effect handlers, it makes sense to extend our language with an additional construct to install an action to run when we *resume* a captured continuation. Figure 15 extends Λ_ρ with a statement for dynamic wind (called Λ_ρ [Eff, Fin, Dyn]). The action s is triggered, whenever the control flow re-enters the region denoted by r.

Example 4.9. Let us look at a larger example in Figure 16, where we use dynamic wind to save the state of an open file upon leaving a region and restore it when we re-enter it. We define a function `withFile` that opens a file and passes a reference to the opened file to the given program `prog`. We install an unwind handler that closes the file whenever control jumps across it, and a rewind handler that reopens the file and seeks to the position where we left off whenever control resumes. If the program `prog` uses a non-determinism effect, for example via `handleForkList`, multiple forks will restart reading from the same file position.

This example does not define a safe API for file access because users of `withFile` have access to the raw file handle. But with regions it is easily possible to define a safe API in terms of `withFile`.

Extended Typing Rules:

$$\frac{\Gamma,\ r,\ l:\ r \sqsubseteq \rho \mid r \vdash\ s\ :\ \tau \qquad \Gamma \mid \rho \vdash\ s\ :\ ()}{\Gamma \mid \rho \vdash\ \textbf{try}\ \{\ [r](l) \Rightarrow s_0\ \}\ \textbf{rewind}\ \{\ s\ \}\ :\ \tau}\ [\text{Rewind}]$$

Extended Translation Rules:

$$\mathcal{S}[\![\ \textbf{try}\ \{\ [r](l) \Rightarrow s_0\ \}\ \textbf{rewind}\ \{\ s\ \}\ ]\!]_\rho\ =\ (\Lambda r \Rightarrow \lambda l \Rightarrow \mathcal{S}[\![\ s_0\ ]\!]_r)\ \ \mathcal{T}[\![\ \rho\ ]\!]\ \ (\text{LiftDyn}\ \mathcal{S}[\![\ s\ ]\!]_\rho)$$

Auxiliary Definitions:

$$\begin{array}{lll} \text{LiftDyn}\ f & : & \forall a.\ \text{Cps}\ R\ a \to \text{Cps}\ R\ a \\ \text{LiftDyn}\ f & = & \Lambda a \Rightarrow \lambda m \Rightarrow \lambda k \Rightarrow m\ (\lambda x \Rightarrow f\ (\lambda u \Rightarrow k\ x)) \end{array}$$

Fig. 15. Extension of $\Lambda_\rho$with dynamic wind (Λ_ρ [Dyn]).

The translation in Figure 15 defines the semantics of the second half of dynamic wind in pure System F. The evidence variable l is bound to a computation that pushes the rewinding statement s onto the continuation. In general, evidence contains both rewinding and unwinding statements. This can also be seen in Example 4.9. Here we pass the evidence $\mathtt{l}_2\ \oplus\ \mathtt{l}_1$, which contains the unwind and rewind code, to `prog`.

While dynamic wind works for resources like files that are recoverable, if we want to combine general control operators like effect handlers (Subsection 4.6) with arena allocation (Subsection 4.2), our conceptual framework highlights a problem: When we deallocate the arena upon leaving the region, there is no way we can recover it upon resumption. Region-based memory management and multi-shot delimited control seem to be fundamentally incompatible. However, based on our computational interpretation of evidence, we can offer safe behavior:

(1) When leaving a region, we can *deallocate* the arena. When we would re-enter the arena's region, we abort with an exception. Since we only throw an exception when we actually try to resume into a region containing a deallocated arena, this makes region-based memory management viable for many programs.
(2) When leaving a region, we can *evacuate* the arena to some garbage collected heap and, for example, add a reference count, or switch to manual memory management. When we re-enter the arena's region, we re-install it from the heap.

Both of these are unsatisfactory for the special case of one-shot continuations. For these, better approaches exist. We leave integrating these into our conceptual framework to future work.

As a final example, the next subsection illustrates the interaction between region-based memory management and first-class coroutines.

4.9 Coroutines

To further investigate the combination of advanced control-flow mechanisms and region-based memory management, we introduce first-class *coroutines*. Coroutines are suspended computations that are either done or can be resumed to yield another coroutine [Moura and Ierusalimschy 2009; Wang and Dahl 1971]. We can implement coroutines in terms of effect handlers, using a recursive data type [Haynes et al. 1986].

```
data Coroutine r t =  Done t | More (Cap r () (Coroutine r t))
```

The common implementation of coroutines as state machines [Bierman et al. 2012] arises from defunctionalization [Danvy and Nielsen 2001] of the stored continuations. There are two interesting kinds of interaction between region-based resource management and first-class coroutines.

```
def withFile[r₀, t](
  prog: [r](GlobalRef File, r ⊑ r₀) →r t
) at r₀ {
  val fileRef = newGlobalRef(openFile("book.txt"));
  val posnRef = newGlobalRef(0);
  try { [r₁](l₁) ⇒
    try { [r₂](l₂) ⇒
      val result = prog[r₂](fileRef, l₂ ⊕ l₁);
      closeFile(getGlobalRef(fileRef));
      return result
    } rewind {
      val file = openFile("book.txt");
      setGlobalRef(fileRef, file);
      seekFile(file, getGlobalRef(posnRef))
    }
  } unwind {
    val file = getGlobalRef(fileRef);
    setGlobalRef(posnRef, filePosn(file));
    closeFile(file)
  }
}
```

Fig. 16. Example program in Λ_ρ [Eff, Fin, Dyn] using dynamic wind.

Example 4.10. First, consider what happens when we allocate a resource *outside* of the coroutine. This example should not and does not typecheck.

```
arena { [r₁](a₁, l₁) ⇒
  val coroutine = try { [r₂](yield, l₂) ⇒
      // use both the arena and the coroutine
      do(yield, (), 𝟘); alloc(a₁, 99, l₂); do(yield, (), 𝟘)
      return (Done(()))
  } with { ((), resume) ⇒ return More(resume) };
  return coroutine // does not typecheck
}
```

The coroutine uses arena a_1. We can not return the coroutine, because it would leave the region where it is safe to resume it.

Example 4.11. Second, consider what happens when we allocate a resource *inside* of the coroutine. Let us assume the following example that defines a coroutine that yields twice.

```
val coroutine = try { [r₁](yield, l₁) ⇒
  do(yield, (), 𝟘);
  arena { [r₂](a₂, l₂) ⇒ do(yield, (), l₂) }
  return (Done(()))
} with { ((), resume) ⇒ return More(resume) };
```

The second yield is performed from within a fresh arena region $\mathtt{r}_2$. Having to use evidence $\mathtt{l}_2$ highlights that for the second yield, and only there, we have to evacuate the arena. Local allocation of arenas are fine, as long as we do not yield out of them.

Again, since coroutines are often resumed exactly once, we would like to add special support for this use case. We could for example require users to manually copy and free suspended coroutines, like in Multicore OCaml [Dolan et al. 2014], which has discontinue and clone primitives for resumptions, or in an extension of Koka [Leijen 2018], which requires manual finalization. Our region-based type system rules out all unsafe combinations of coroutines with resources, while allowing for a large number of safe uses.

4.10 Summary

In this section, we have extended our core language Λ_ρ with numerous language features, such as exceptions, local state, effect handlers, and dynamic wind. We uniformly presented the semantics of those features as translations to pure System F, enabling a well-defined composition into larger languages and allowing us to study interactions between the features.

5 RELATED WORK

Out of the numerous works about regions for resource management, the one most closely related, and indeed which has been the basis of our work, is [Kiselyov and Shan 2008], which in turn is based on [Fluet and Morrisett 2004]. Kiselyov and Shan provide a library for region-based resource management in Haskell. They also propose to use regions for resources other than memory, introduce explicit subregioning witnesses, and properly handle builtin Haskell exceptions. They demonstrate how types and regions are inferred, which we do not discuss. Their approach perfectly fits into our conceptual framework, and allows us to extend their approach and discuss control effects beyond handling of builtin exceptions.

The non-trivial operational interaction between operators for delimited control and dynamic binding, or more general continuation marks, has been discussed before [Flatt and Dybvig 2020; Flatt et al. 2007; Kiselyov et al. 2006]. Our conceptual framework supports these use cases and cleanly connects the type level and the operational semantics.

Makholm [2000] and Phan et al. [2008] discuss region-based memory management in the logic programming languages Prolog and Mercury respectively. Harris [2005] discuss the interaction between exceptions and atomic blocks in software transactional memory in Java. It would be interesting to see if the challenges that they address can be cast into our conceptual framework and if we could reproduce their solutions.

Our treatment of effect handlers in Section 4.6 follows recent developments in type systems for effect handlers [Biernacki et al. 2019b; Brachthäuser et al. 2020b; Zhang and Myers 2019; Zhang et al. 2020]. Our evidence terms are a generalization of the evidence vectors proposed by Xie et al. [2020] and our formal treatment of the operational semantics is inspired by their concept of evidence correspondence. They introduce the property of scoped resumptions and dynamically check that it holds. We statically enforce this property.

Our CPS-based semantics of exceptions, our control operator, and effect handlers is closely related to the one presented by Schuster et al. [2020]. However, they do not support effect-polymorphic functions, which can be expressed within our framework. Our translation of effect handlers to System F is similar to the one given in Appendix B of [Hillerström et al. 2017].

Kiselyov and Ishii [2015] 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 works in the presence of an exception effect. Other effects, like non-determinism, are explicitly ruled out by the type system when they would be used across a region. The biggest difference to our work is that they reify the structure of the program as a free monad and then write interpreters over this structure, whereas we translate programs to iterated CPS. They understand regions as effects, we understand regions as parts of the stack and, at least for the case of arenas with exceptions, we provide a proof of safety. A more minor difference is that we pass arenas explicitly while they index nested regions by a type-level natural number for disambiguation.

Leijen [2018] reports on an extension of the programming language Koka with support for resources and finalization. This 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.

Ahman and Bauer [2020] present another approach to resources in the presence of algebraic effects and handlers: Runners. They guarantee that finalizers are run exactly once by requiring runners to always resume exactly once. It is very useful to have this guarantee. If the only control effect are exceptions, we offer the same guarantee, but we do not prove it. We go further and discuss arbitrary control effects with arbitrary nesting where we cannot offer such a guarantee. We present an operational semantics that relates resource management to the stack and a denotational semantics that translates programs to iterated CPS. Their denotational semantics translates programs to essentially a free monad.

6 CONCLUSION

We presented a unified treatment of region-based resource management and control effects in a language with types and effects. It rests on the central idea that a region denotes a part of the runtime stack. We have formalized the connection between type-level regions and the actual shape of the runtime stack during evaluation. We have demonstrated that our conceptual framework can incorporate a large number of different language features and discussed their non-trivial interaction. All of these features are bound together by a translation to continuation-passing style and a denotational interpretation of evidence as answer type coercions, which further emphasizes the understanding of regions as a property of the runtime context.

However, there are a couple of remaining challenges that we leave to future work. The language we presented cannot support more exotic uses of delimited control, like for example dynamically overwriting an exception handler after capturing the continuation. This changes the region the next statement runs in, so we need region modification which is exactly answer-type modification in our CPS-based semantics. Moreover, while the interaction between finalizers and multiple invocations of the current continuation is problematic, linear use of the continuation is unproblematic. Currently we have no special provisions to incorporate this knowledge.

Our conceptual framework, which already supports many uses of delimited control, will form the foundation upon which we will build these future investigations.

REFERENCES

Danel Ahman and Andrej Bauer. 2020. Runners in Action. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 29–55.

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

Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause'n'Play: Formalizing Asynchronous C#. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 233–257.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019a. Abstracting Algebraic Effects. *Proc. ACM Program. Lang.* 3, POPL, Article 6 (Jan. 2019), 28 pages.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019b. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371116

Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: Extensible Algebraic Effects in Scala (Short Paper). In *Proceedings of the International Symposium on Scala* (Vancouver, BC, Canada). ACM, New York, NY, USA. https://doi.org/10.1145/3136000.3136007

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect Handlers for the Masses. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 111 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276481

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). https://doi.org/10.1145/3428194

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. Effekt: Capability-Passing Style for Type- and Effect-safe, Extensible Effect Handlers in Scala. *Journal of Functional Programming* (2020). https://doi.org/10.1017/S0956796820000027

Edwin Brady. 2020. *Idris 2: Quantitative Type Theory in Action.* Technical Report. University of St Andrews, Scotland, UK. https://www.type-driven.org.uk/edwinb/papers/idris2.pdf

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with Continuations, or Without? Whatever. *Proc. ACM Program. Lang.* 3, ICFP, Article 79 (July 2019), 28 pages. https://doi.org/10.1145/3341643

Olivier Danvy. 2004. On Evaluation Contexts, Continuations, and the Rest of Computation. (02 2004).

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the Conference on LISP and Functional Programming* (Nice, France). ACM, New York, NY, USA, 151–160.

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the Conference on Principles and Practice of Declarative Programming* (Florence, Italy). 162–174.

Stephen Dolan, Leo White, and Anil Madhavapeddy. 2014. Multicore OCaml. In *OCaml Workshop*.

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

Matthias Felleisen. 1988. The Theory and Practice of First-class Prompts. In *Proceedings of the Symposium on Principles of Programming Languages* (San Diego, California, USA). ACM, New York, NY, USA, 180–190.

Matthew Flatt and R. Kent Dybvig. 2020. Compiler and Runtime Support for Continuation Marks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/3385412.3385981

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment. In *Proceedings of the International Conference on Functional Programming* (Freiburg, Germany). Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/1291151.1291178

Matthew Fluet and Greg Morrisett. 2004. Monadic Regions. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/1016850.1016867

Daniel P. Friedman and Christopher T. Haynes. 1985. Constraining Control. In *Proceedings of the Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA). ACM, 245–254.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

Tim Harris. 2005. Exceptions and side-effects in atomic blocks. *Science of Computer Programming* 58, 3 (2005), 325 – 343. https://doi.org/10.1016/j.scico.2005.03.005 Special Issue on Concurrency and synchonization in Java programs.

Christopher T Haynes. 1987. Logic continuations. *The Journal of Logic Programming* 4, 2 (1987), 157–176.

Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. 1986. Obtaining coroutines from continuations. *Computer languages* 11, 3-4 (1986), 143–153.

Robert Hieb and R. Kent Dybvig. 1990. Continuations and Concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (Seattle, Washington, USA) *(PPOPP '90)*. ACM, New York, NY, USA, 128–136.

Daniel Hillerström, Sam Lindley, Bob Atkey, and KC Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *Formal Structures for Computation and Deduction (LIPIcs, Vol. 84)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

R.John Muir Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Inform. Process. Lett.* 22, 3 (1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the International Conference on Functional Programming* (Freiburg, Germany). ACM, New York, NY, USA, 177–190.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the Haskell Symposium* (Vancouver, BC, Canada). ACM, New York, NY, USA, 94–105.

Oleg Kiselyov and Chung-chieh Shan. 2008. Lightweight Monadic Regions. In *Proceedings of the Haskell Symposium* (Victoria, BC, Canada) *(Haskell '08)*. ACM, New York, NY, USA.

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited Dynamic Binding. In *Proceedings of the International Conference on Functional Programming* (Portland, Oregon, USA). ACM, New York, NY, USA, 26–37.

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/178243.178246

Daan Leijen. 2017. Structured Asynchrony with Algebraic Effects. In *Proceedings of the Workshop on Type-Driven Development* (Oxford, UK). ACM, New York, NY, USA, 16–29.

Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report MSR-TR-2018-10. Microsoft Research. 35 pages.

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210.

Henning Makholm. 2000. A Region-Based Memory Manager for Prolog. In *Proceedings of the 2nd International Symposium on Memory Management* (Minneapolis, Minnesota, USA) *(ISMM '00)*. Association for Computing Machinery, New York, NY, USA, 25–34. https://doi.org/10.1145/362422.362434

Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (Feb. 2009), 31 pages.

Koen Pauwels, Tom Schrijvers, and Shin-Cheng Mu. 2019. Handling Local State with Global State. In *Proceedings of Mathematics of Program Construction (MPC)*. Springer.

Quan Phan, Zoltan Somogyi, and Gerda Janssens. 2008. Runtime Support for Region-Based Memory Management in Mercury. In *Proceedings of the 7th International Symposium on Memory Management* (Tucson, AZ, USA) *(ISMM '08)*. Association for Computing Machinery, New York, NY, USA, 61–70. https://doi.org/10.1145/1375634.1375644

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM annual conference* (Boston, Massachusetts, USA). ACM, New York, NY, USA, 717–740.

Philipp Schuster and Jonathan Immanuel Brachthäuser. 2018. Typing, Representing, and Abstracting Control. In *Proceedings of the Workshop on Type-Driven Development* (St. Louis, Missouri, USA). ACM, New York, NY, USA, 14–24. https://doi.org/10.1145/3240719.3241788

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4, ICFP, Article 93 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408975

Dorai Sitaram and Matthias Felleisen. 1990. Control delimiters and their hierarchies. *LISP and Symbolic Computation* 3, 1 (01 Jan 1990), 67–99.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST. *Proc. ACM Program. Lang.* 2, POPL, Article 64 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158152

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, and Peter Sestoft. 2001. Programming with Regions in the ML Kit (for Version 4). (10 2001).

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value λ-Calculus Using a Stack of Regions. In *Proceedings of the Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. ACM, New York, NY, USA, 188–201. https://doi.org/10.1145/174675.177855

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Arne Wang and O. Dahl. 1971. Coroutine sequencing in a block structured environment. *BIT Numerical Mathematics* 11 (1971), 425–449.

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. *Proc. ACM Program. Lang.* 4, ICFP, Article 99 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408981

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages.

Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting Blame for Safe Tunneled Exceptions. In *Proceedings of the Conference on Programming Language Design and Implementation* (Santa

Barbara, CA, USA). ACM, New York, NY, USA, 281–295.

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling Bidirectional Control Flow. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 139 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428207