From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers

Marius Müller
University of Tübingen, Germany

Philipp Schuster
University of Tübingen, Germany

Jonathan Lindegaard Starup
Aarhus University, Denmark

Klaus Ostermann
University of Tübingen, Germany

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Abstract
Effect handlers are a high-level abstraction that enables programmers to use effects in a structured way. They have gained a lot of popularity within academia and subsequently also in industry. However, the abstraction often comes with a significant runtime cost and there has been intensive research recently on how to reduce this price.

A promising approach in this regard is to implement effect handlers using a CPS translation and to provide sufficient information about the nesting of handlers. With this information the CPS translation can decide how effects have to be lifted through handlers, *i.e.*, which handlers need to be skipped, in order to handle the effect at the correct place. A structured way to make this information available is to use a calculus with a region system and explicit subregion evidence. Such calculi, however, are quite verbose, which makes them impractical to use as a source-level language.

We present a method to infer the lifting information for a calculus underlying a source-level language. This calculus uses second-class capabilities for the safe use of effects. To do so, we define a typed translation to a calculus with regions and evidence and we show that this lift-inference translation is typability- and semantics-preserving. On the one hand, this exposes the precise relation between the second-class property and the structure given by regions. On the other hand, it closes a gap in a compiler pipeline enabling efficient compilation of the source-level language. We have implemented lift inference in this compiler pipeline and conducted benchmarks which indicate that the approach is indeed working.

Main Reference Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. Proc. ACM Program. Lang. 7, OOPSLA2, Article 255 (October 2023), 29 pages. https://doi.org/10.1145/3622831

Comments This is an extended version of the main reference. Compared to the published paper, this report contains the full appendix: detailed proof of typability preservation of the lift-inference translation, proof sketch of semantics preservation of the lift-inference translation, formal definition of the abstract machine for System $\Xi/\Lambda_{\mathsf{cap}}$, full definition of the CPS translation, detailed proof of the semantics preservation of the CPS translation and changes necessary for the proof of typability preservation of the CPS translation relative to the original version.

1 Introduction

Languages with effect handlers [29, 30] offer a high-level way to structure effectful programs. Effect handlers allow for a combination of various effects by giving meaning to abstract effect operations (such as exceptions, async-await, generators, logic programming, or probabilistic programming) in a composable way. In the past decade, effect handlers have been a hot topic in programming language research and have also gained more and more popularity outside academia. They have been implemented not only in research languages (such as Eff [1], Koka [21], Frank [24], Effekt [4], or Helium [3]), but also practical general purpose languages such as OCaml [38], Scala[2], Unison[3], and WebAssembly[1] are following lead and have started to integrate effects and handlers. However, to make effect handlers practically useful, it is critically important to minimize the runtime cost incurred by this abstraction.

Our goal, thus, is to have a source-level language with effect handlers that can be compiled efficiently. Ideally, the language should be effect-safe and sufficiently expressive without being unnecessarily complex. A recently developed way to strike this balance is using lightweight effect polymorphism. Brachthäuser et al. [4] show how to design such a system using *second-class capabilities* [28]. They develop the language Effekt which features lexical effect handlers [4, 45, 3], a recent variant of effect handlers, and show how to translate it to System Ξ, a calculus in explicit capability-passing style. The translation preserves typing and is used to give the semantics for Effekt.

A recently developed way to efficiently implement lexical effect handlers uses iterated continuation-passing style (CPS). It has been shown to yield good performance results [35]. Moreover, Schuster et al. [36] have designed a core calculus Λ_{cap}, which features effect handlers based on *regions*. They show how to translate Λ_{cap} to pure System F in a typability- and semantics-preserving way.

Figure 1 summarizes the developments mentioned above. To reach our goal and complete the pipeline, we have to close the gap in the middle: we have to show how to translate from a system with second-class capabilities to a system with region-based effects, *i.e.*, we have to understand the relation between these two concepts precisely.

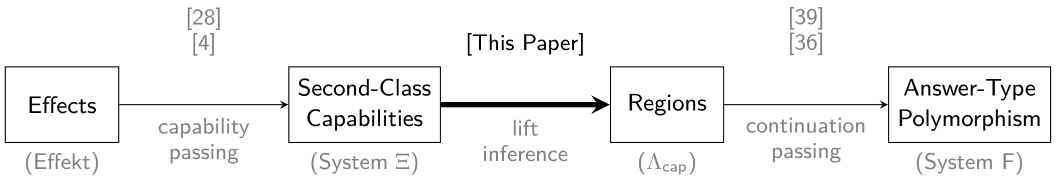

Figure 1 Overview of this paper in relation to prior work. Nodes are labeled with mechanisms to ensure effect safety (e.g., Effects); below each node one example calculus is listed. Each arrow corresponds to a translation between calculi.

To do so, in this paper we present a typed translation from the calculus of second-class capabilities System Ξ to the region-based calculus Λ_{cap}. The typed nature of the translation makes the relation between the two concepts explicit. It connects the lexical scopes of the definition sites of capabilities in System Ξ with corresponding regions in Λ_{cap}, in which the capabilities are allowed to be used. This connection also materializes in the definition of sound translation environments (see Definition 4), which are maintained during the translation as a key component.

To practically evaluate our approach, we have implemented the translation as a compiler phase in the Effekt language. As our implementation strategy is to translate effects and handlers to CPS [14, 35, 36], we have also implemented the CPS translation of Schuster et al. [36] with Standard ML [26] as the target language, hence completing a compiler pipeline for Effekt. To further compile SML we use the MLton[4] compiler.

[2] https://github.com/lampepfl/dotty/pull/16739
[3] https://www.unison-lang.org/learn/fundamentals/abilities
[1] https://github.com/effect-handlers/wasm-spec
[4] http://mlton.org

For a certain restricted class of programs Schuster et al. [35] prove that all abstractions related to effects and handlers can theoretically be eliminated. Moreover, they demonstrate excellent performance on a number of benchmarks which have independently been reproduced by Karachalias et al. [18]. However, for their performance evaluation, they wrote programs directly in a core language λ_{cap}. We, for the first time, can reproduce the performance claims of Schuster et al. [35] in a realistic source-level language.

Comparing our implementation with other state-of-the-art implementations of effect handlers, we found that our relative performance ranges from 2.1x slowdown to 44.4x speedup compared with OCaml [38], 1.1x slowdown to 87.8x speedup compared with Koka [21, 44], and 1.2x slowdown to 23.3x speedup compared with Eff [31, 18]. The results indicate that the compilation technique presented by Schuster et al. [35] works for a high-level language presented by Brachthäuser et al. [4] and yields good performance.

By closing the conceptual gap between lexical scoping and regions, we enable efficient compilation of lexical effect handlers, like those found in Effekt or Helium. However, our results do not immediately carry over to dynamic effect handlers, as implemented in OCaml 5, Koka, and WebAssembly and we leave further investigation of this to future work.

Our contributions are the following.

- We formally present a typability- and semantics-preserving translation from System Ξ to Λ_{cap}. We refer to this as *lift-inference translation* for reasons to be explained shortly.
- From a theoretical perspective, we hence clarify the precise relation between scope-based reasoning for second-class capabilities and region-based reasoning.
- From a practical perspective, we fill in an important missing link in the compiler pipeline illustrated in Figure 1.
- In Λ_{cap}, continuation calls have to be scoped [44]. No such restriction exists for System Ξ. To support System Ξ, we lift the scoped-continuation restriction imposed by Schuster et al. [36] and thus generalize Λ_{cap}. We prove that the generalized system still allows for a translation to iterated CPS satisfying the same properties as the original.
- We have implemented the lift-inference translation and the CPS translation as steps for the efficient compilation of the source-level language Effekt to SML. In addition, we have performed benchmarks indicating that our approach is competitive with or often faster than other state-of-the-art languages featuring effect handlers.

Next, in Section 2, we introduce the main ideas of the lift-inference translation by considering examples. In Section 3, we first recap the two calculi involved and then formally present the lift-inference translation. A discussion of the implementation and the corresponding benchmarks is given in Section 4. In Section 5, we compare to related work. We conclude and outline future work in Section 6.

2 Main Ideas

In this section we introduce the main ideas for our lift-inference translation from the source calculus System Ξ [4] to the target calculus Λ_{cap} [36]. Both calculi are mostly standard functional languages with multi-arity functions, but additionally feature lexical effect handlers [4, 45, 3]. Effect handlers are called *lexical* if effect operation calls are lexically related to the corresponding effect handler. This is in contrast to the more traditional dynamically scoped handlers, which search the stack for the closest handler at runtime. Both calculi establish this lexical connection between effect and handler by using explicit capability-passing style

```
try { (yield_1) ⇒
  def g(i : Int) { do yield_1(i) };
  val x = g(1);
  try { (yield_2) ⇒
    g(x)
  } with { (j, k) ⇒ 42 }
} with { (j, k) ⇒ do k(j + 1) }
```

(a) Program in System Ξ. Effect safety is established by treating capabilities as second-class values.

```
try { [r_1; n_1 : r_1 ⊑ ⊤](yield_1) ⇒
  def g[r; n : r ⊑ r_1](i : Int) at r { do yield_1[n](i) };
  val x = g[r_1; 𝟘](1);
  try { [r_2; n_2 : r_2 ⊑ r_1](yield_2) ⇒
    g[r_2; n_2](x)
  } with { (j, k) ⇒ 42 }
} with { (j, k) ⇒ do k[𝟘](j + 1) }
```

(b) Program in Λ_{cap}. Effect safety is established by tracking the region of a capability and requiring subregion evidence.

Figure 2 Simple example illustrating lexical effect handling.

[4, 45]. This means that effects are bound as capabilities by their handlers and can then be used in the scope of the handled statement.

Both languages are effect safe, that is, they guarantee that all effect operations are eventually handled. In System Ξ, this is achieved by making functions and capabilities *second-class* [28] so that they cannot be returned or stored in data structures, hence making sure that capabilities cannot leave their defining scope. To make this explicit, these second-class functions and capabilities are called blocks. While blocks in System Ξ are required to be second-class, they still can be higher-order, *i.e.*, functions can abstract block parameters.

In Λ_{cap}, there is no such second-class restriction on functions and capabilities. To ensure effect safety, Λ_{cap} features a region system with subregioning and explicit subregion evidence instead. Here, a *region* denotes the scope of an effect handler and subregion evidence is used to constructively witness how handlers are nested. By enforcing that capabilities can only be called in a subregion of their corresponding handler, this system also ensures that they cannot leave their defining scope.

Since evidence terms precisely witness how handlers are nested, they contain the information of where capabilities have to be *lifted* to when they are called, *i.e.*, how many handlers have to be jumped over until the correct handler is found. This enables efficient compilation of effects and handlers [36, 35]: the lifting can often be promoted to compile time which avoids the search for the correct handler at runtime. The goal of our lift-inference translation is thus to infer this lifting information by endowing terms in System Ξ with correct regions and evidence to obtain valid terms in Λ_{cap}.

2.1 Basic Example

To illustrate the need for lifting, as well as how to perform lift inference, consider the following example in Effekt:

```
effect Yield(i: Int): Int
try {
  def g(i: Int): Int / {} = { do Yield(i) };
  val x = g(1);
  try { g(x) } with Yield { j ⇒ 42 }
} with Yield { j ⇒ resume(j + 1) }
```

We define a local function g, which uses the Yield effect to return an integer. It is annotated to have type Int and no observable effects {}. This means the (dynamic) call site of g cannot handle the Yield effect and it needs to be handled at the (lexical) definition site of g. In consequence, running the example will result in the integer 3.

Subfigure 2 (a) shows the result of the type-and-effect directed translation [4] from Effekt (with support for effect inference) to System Ξ in explicit capability-passing style. Comparing the System Ξ term to the original program, we notice that handlers explicitly bind capabilities (*e.g.*, yield_1) and effect calls now directly refer to a capability (*e.g.*, $\mathbf{do}\ \mathsf{yield}_1(\mathsf{i})$). Capability passing in System Ξ also makes explicit that the effect call in the body of g refers to the outer handler.

At runtime, we need to make sure to handle the effect operation with the correct handler. Thus, when g is called the second time, we have to lift the capability in its body through the inner handler, that is, we need to skip over the inner handler to transfer control flow to the outer handler. Our goal is to make this skipping of handlers explicit. Then, our CPS translation can make use of this information to avoid searching for the correct handler at runtime, as it knows how many segments of the stack it has to capture.

One possibility, to make this information explicit, would be to use lifting annotations [35, 2]. The definition of g (for the second call) would then become

$$\mathbf{def}\ \mathsf{g}(\mathsf{i} : \mathsf{Int})\ \{\ \mathbf{do}\ (\mathbf{lift}\ \mathsf{yield}_1)(\mathsf{i})\ \};$$

But this is only correct when g is called under the inner handler. When it is called the first time, immediately after its definition, this annotation is incorrect since no handler has to be skipped. It is hence not clear what lifting annnotation should be used when defining g. The lifting information at the definition-site should be correct for any call-site.

A very structured and sufficiently expressive way to deal with this situation is to use regions and subregion evidence instead. They allow us encode the lifting information and also give us the ability to abstract over it at the definition-site. This can be seen in Subfigure 2 (b), which shows what the example looks like in Λ_{cap}. Each handler now not only binds a capability, but also a fresh region (*e.g.*, r_1) and subregion evidence (*e.g.*, $\mathsf{n}_1 : \mathsf{r}_1 \sqsubseteq \top$) witnessing that the fresh region is a subregion of the current one (*e.g.*, the toplevel region $\top$). The basic idea now is to abstract the required lifting information in the form of an evidence parameter n for g, which is then provided to the call of the capability yield_1 in the function body. Capability yield_1 is bound at the outer handler in region r_1, so its evidence should witness that the region in which yield_1 is called is a subregion of r_1 and which subregion it is. To express this, we also abstract over a region parameter r for g, which stands for the region at its call-site as is visible in the annotation $\mathbf{at}\ \mathsf{r}$. The evidence parameter n is thus typed as $\mathsf{r} \sqsubseteq \mathsf{r}_1$, expressing that the call-site region r must be a subregion of g's (and yield_1's) definition region r_1.

When calling g under the inner handler, the current region is the region r_2 bound at this handler, so we instantiate g's region parameter with r_2. The evidence passed to g thus must have type $\mathsf{r}_2 \sqsubseteq \mathsf{r}_1$. This subregion relation is witnessed by evidence n_2 bound at the inner handler. But now we can also call g immediately after defining it, in which case we instantiate its region parameter with r_1. For the evidence we then use the trivial one, $\mathbb{0} : \mathsf{r}_1 \sqsubseteq \mathsf{r}_1$, stating that subregioning is reflexive. In in either case, the evidence passed to yield_1 correctly witnesses how the capability must be lifted.

To see how this lifting works, consider the CPS translation of the above program:

$$
\begin{array}{l}
\textsc{Reset}(\\
\quad (\Lambda \mathsf{r}_1.\ \lambda \mathsf{n}_1.\ \lambda \mathsf{yield}_1. \\
\qquad \lambda \mathsf{kg}.\ (\lambda \mathsf{k}.\ \mathsf{k}\ (\Lambda \mathsf{r}.\ \lambda \mathsf{n}.\ \lambda \mathsf{i}.\ \mathsf{n}\ \mathsf{Int}\ (\mathsf{yield}_1\ \mathsf{i}))) \\
\qquad\qquad (\lambda \mathsf{g}.\ (\lambda \mathsf{k}_1.\ (\mathsf{g}\ \mathsf{r}_1\ (\Lambda \mathsf{a}.\ \lambda \mathsf{m}.\ \mathsf{m})\ 1) \\
\qquad\qquad\qquad\qquad (\lambda \mathsf{x}.\ \textsc{Reset}(\\
\qquad\qquad\qquad\qquad\qquad (\Lambda \mathsf{r}_2.\ \lambda \mathsf{n}_2.\ \lambda \mathsf{yield}_2.\ \mathsf{g}\ \mathsf{r}_2\ \mathsf{n}_2\ \mathsf{x}) \\
\qquad\qquad\qquad\qquad\qquad (\textsc{Cps}\ \mathsf{r}_1\ \mathsf{Int}) \\
\qquad\qquad\qquad\qquad\qquad \textsc{Lift} \\
\qquad\qquad\qquad\qquad\qquad (\lambda \mathsf{j}.\ \lambda \mathsf{k}.\ \lambda \mathsf{k}_0.\ \mathsf{k}_0\ 42) \\
\qquad\qquad\qquad\qquad\quad) \\
\qquad\qquad\qquad\qquad\quad \mathsf{k}_1)) \\
\qquad\qquad\quad \mathsf{kg})) \\
\quad (\textsc{Cps}\ \mathsf{Void}\ \mathsf{Int}) \\
\quad \textsc{Lift} \\
\quad (\lambda \mathsf{j}.\ \lambda \mathsf{k}.\ (\Lambda \mathsf{a}.\ \lambda \mathsf{m}.\ \mathsf{m})\ \mathsf{Int}\ (\lambda \mathsf{k}_0.\ \mathsf{k}\ (\mathsf{j}\ +\ 1)\ \mathsf{k}_0)) \\
)
\end{array}
$$

Here we can see that the evidence parameters n_1, n_2 bound at the handlers are eventually instantiated with the function LIFT which has the effect of capturing a further delimited continuation. The delimiters for continuations are given by the meta function RESET which is wrapped around each handler upon translation and has the effect of applying its argument to an empty continuation. For the definition of LIFT and RESET we refer to Subsection 3.3. The trivial evidence $\mathbb{0}$ just becomes the (polymorphic) identity function, $\Lambda \mathsf{a}.\ \lambda \mathsf{m}.\ \mathsf{m}$.

The function g is translated to

$$\Lambda \mathsf{r}.\ \lambda \mathsf{n}.\ \lambda \mathsf{i}.\ \mathsf{n}\ \mathsf{Int}\ (\mathsf{yield}_1\ \mathsf{i})$$

In the function body we can see that the application of the capability is translated to an application of the evidence parameter n to this capability. When looking at the call sites we can then see how the lifting happens concretely. At the outer call site, n is instatiated with the identity function so that no lifting happens. At the inner call site (under the second RESET), n is instantiated with n_2, *i.e.*, with LIFT, so that here the delimited continuation which is captured does not end at the inner RESET but at the outer one.

To sum up this subsection, after lift inference each function-block definition should abstract a fresh region standing for the region in which the function will run, and an evidence parameter witnessing that this call-site region is a subregion of the function's definition-site region.

2.2 Higher-Order Functions

The example in the previous subsection only uses first-order functions. However, System Ξ also supports higher-order functions which make lift inference a bit more complicated. To see why, consider the following variation of the example from the previous subsection:

```
def call { f: Int ⇒ Int / {} } =
  val x = f(1);
  try { f(x) }
  with Yield { j ⇒ 42 }
```

```
try {
  call { (i: Int) ⇒ do Yield(i) }
} with Yield { j ⇒ resume(j + 1) }
```

As in the previous example, the effect operation is called under two handlers and the result of running it is the same. This time however, the inner handler is installed by a higher-order function `call`. This example motivates, why lexical effect handling can be desirable: as programmers, we want to reason locally about the relation of the call to `do Yield` and its

```
def call(f : Int → Int) {
  val x = f(1);
  try { (yield₂) ⇒ f(x) }
  with { (j, k) ⇒ 42 }
};

try { (yield₁) ⇒
  call { (i : Int) ⇒ do yield₁(i) }
} with { (j, k) ⇒ do k(j + 1) }
```

(a) Program in System Ξ.

```
def call[r_c, r_f; n_c : r_c ⊑ ⊤, n_f : r_c ⊑ r_f](
    f : ∀[r; r ⊑ r_f](Int) →r Int
  ) at r_c {
    val x = f[r_c; n_f](1);
    try { [r₂; n₂ : r₂ ⊑ r_c](yield₂) ⇒ f[r₂; n₂ ⊕ n_f](x) }
    with { (j, k) ⇒ 42 }
};

try { [r₁; n₁ : r₁ ⊑ ⊤](yield₁) ⇒
  call[r₁, r₁; n₁, 𝟘] { [r_g; n_g](i : Int) at r_g ⇒ do yield₁[n_g](i) }
} with { (j, k) ⇒ do k[𝟘](j + 1) }
```

(b) Program in Λ_{cap}.

Figure 3 Example with higher-order function.

lexically enclosing handler, without having to be aware of `call`'s implementation details.

Subfigure 3 (a) shows the program in System Ξ. Again, the capability-passing translation makes the lexical relation of the effect call and the outer handler explicit.

The translation to Λ_{cap} in Subfigure 3 (b) is now slightly more involved. Handlers are endowed with regions and evidence as before. The block passed to call is the same as g in the previous example, it is just anonymous now. Its translation thus is the same as for g, it abstracts a fresh region r_g and evidence $n_g : r_g \sqsubseteq r_1$ which it then passes to the capability in its body.

As the anonymous block is called indirectly via the parameter f of function call, we have to pass an appropriate region and evidence to f in the body of call. Since this region and evidence should be correct for any block f is instantiated with, we abstract over them in the definition of call. This way, we can provide them at the call-site of call when we know the concrete block argument we pass for f.

Therefore, call now abstracts two regions and two evidence parameters. Region r_c again stands for the region where call will run later and evidence n_c again witnesses that r_c is a subregion of call's definition region[5]. The second region r_f represents the definition region of the block argument passed for f. The second evidence n_f witnesses that r_c is a subregion of r_f. Moreover, in the type of f we can see that, as any other function block, f abstracts a fresh region r and corresponding evidence witnessing that r is a subregion of r_f. When f is called the first time in the body of call, immediately at the beginning, the current region is r_c, so we instantiate the region parameter of f with this region. The evidence for f hence has to witness the subregion relation $r_c \sqsubseteq r_f$ which is precisely what the evidence n_f specifically abstracted for f does. When f is called the second time under the second handler, the current region is not r_c anymore but the region r_2 abstracted at that handler, so we instantiate the region parameter of f with r_2. To obtain the correct evidence for f we thus have to compose the evidence n_f with evidence witnessing that $r_2 \sqsubseteq r_c$. This relation is precisely witnessed by the evidence n_2 abstracted at the handler. The evidence passed to the second call of f thus is the composition $n_2 \oplus n_f$. In the CPS translation this composition of evidence becomes function composition, so that multiple Lift functions can be combined to obtain the overall

[5] For call, this is the toplevel region $\top$ of which any region is a subregion, so n_c is not really necessary in this case.

lift if necessary.

In the application of call, we have to provide appropriate regions and evidence. The first region argument is again the current region, which is now r_1, and the first evidence is n_1, because call was defined outside of the handler. The second region parameter must be instantiated with the definition-site region of the block argument, which is again r_1, as the anonymous block is defined in place. The second evidence hence has to witness that $\mathsf{r}_1 \sqsubseteq \mathsf{r}_1$, so we have to pass the trivial evidence $\mathbb{0}$. Note that in call this evidence is then passed to the calls of the block argument and eventually to yield_1, where for the second call it is composed with n_2 beforehand. Hence, yield_1 indeed receives the correct evidence in both cases since composing with the trivial evidence eventually has no effect. In the CPS translation it is just composition with the identity function.

In general, after lift inference each function block should abstract an additional region and evidence parameter for each of its block parameters. The region stands for the definition-site region of the block argument and the evidence witnesses that the region in which the whole function block is called must be a subregion of the region for the block parameter.

2.3 Summary

Summing up, the guiding principle of our lift-inference translation is that each call-site region of a block is a subregion of its definition-site region. This is facilitated by the second-class property of blocks in System Ξ. Every function block thus abstracts a fresh region in which it will run later and evidence witnessing the above principle. When a block is called, its region parameter is instantiated with the current region and its evidence parameter with appropriate evidence. Hence, we have to track the current region during the translation and we have to remember the regions in which the blocks have been defined. We also have to keep track of the correct evidence for each block.

Moreover, for each block parameter of a function an additional region and evidence parameter is abstracted. The region represents the definition-site of the instantiation of the block parameter and the evidence witnesses that the whole function is called in a subregion of that definition-site region. As the block parameter cannot be returned, this ensures that its instantiation again satisfies our guiding principle.

3 Technical Development

In this section, we formally present how the lift-inference translation from our version of System Ξ to our version of Λ_{cap} proceeds. This translation infers correct regions and evidence for a well-typed term in System Ξ to yield a well-typed term in Λ_{cap}. Before doing so, we recap both languages and detail the changes we have made relative to the original versions of the languages to overcome technical difficulties.

3.1 Syntax and Type Systems

We first describe the syntax and type systems of the two calculi. In the following, source calculus means System Ξ, not to be confused with the source-level language Effekt it underlies.

3.1.1 Source Calculus System Ξ

We start with the source calculus System Ξ, a calculus with lexical effect handlers in explicit capability-passing style with second-class functions.

Syntax of Terms:

Statements

s	$::=$	$\textbf{val}\ x\ =\ s;\ s$	sequencing
	$\mid$	$\textbf{return}\ v$	returning values
	$\mid$	$\textbf{def}\ f\ =\ b;\ s$	defining blocks
	$\mid$	$b(\overline{v},\ \overline{b})$	calling blocks
	$\mid$	$\textbf{do}\ b(v)$	performing capabilities
	$\mid$	$\textbf{try}\ \{\ (c) \Rightarrow s\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}$	handling effects

Values

v	$::=$	x	variables
		$\mid () \mid 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \ldots$	constants

Blocks

b	$::=$	$f,\ k,\ c \mid w$

Block Values

w	$::=$	$\{\ (\overline{x:\ \tau},\ \overline{f:\ \sigma}) \Rightarrow s\ \}$

Syntax of Types:

Value Types

τ	$::=$	$\mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \ldots$

Block Types

σ	$::=$	$(\overline{\tau},\ \overline{\sigma}) \to \tau$	functions
	$\mid$	$\textbf{Cap}\ \tau\ \tau$	capabilities

Environments:

Value Environment

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

Block Environment

Δ	$::=$	$\emptyset \mid \Delta,\ f:\ \sigma$

Names:

Value Variables $x,\ y\ \in \mathsf{x},\ \mathsf{y}$ Block Variables $f,\ g,\ k,\ c\ \in \mathsf{f},\ \mathsf{g},\ \mathsf{k},\ \mathsf{Fail},\ \mathsf{Choice},\ \ldots$

Figure 4 Syntax of System Ξ.

Syntax The syntax of System Ξ is given in Figure 4. The calculus syntactically distinguishes potentially effectful statements from terms that cannot have control effects, *i.e.*, it is in fine-grain call-by-value [22]. Non-effectful terms are further divided into values and blocks. Values are either variables or constants, blocks are either variables or anonymous multi-arity functions. It is important to note that only values can be returned but blocks cannot, that is, blocks are second class. Still, blocks can be higher-order, *i.e.*, they cannot only abstract value parameters but also block parameters.

Statements can be sequenced using $\textbf{val}\ x\ =\ s_1;\ s_2$ where the result of s_1 is bound to variable x in s_2. Defining a local block is done with $\textbf{def}\ f\ =\ b;\ s$ making block b available in the scope of statement s by binding it to variable f. We distinguish calls of a function block from calling capabilities standing for effect operations. For the latter we add the construct $\textbf{do}\ b(v)$ and also reflect this on the type level by adding an additional block type for capabilities. Syntactically distinguishing capabilities is a minor technical difference to the original version of System Ξ, but it simplifies the presentation of the lift-inference translation

Value Typing

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

$$\frac{\Gamma(x) \ = \ \tau}{\Gamma \vdash \ x \ : \ \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash \ 3 \ : \ \textsf{Int}}\ [\textsc{Lit}]$$

Block Typing

$$\boxed{\begin{array}{ccccc}\Gamma \mid & \Delta \vdash & b & : & \sigma \\ \uparrow & \uparrow & \uparrow & & \downarrow\end{array}}$$

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

Statement Typing

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

$$\frac{\Gamma \mid \Delta \vdash \ s_0 \ : \ \tau_0 \qquad \Gamma, \ x : \ \tau_0 \mid \Delta \vdash \ s \ : \ \tau}{\Gamma \mid \Delta \vdash \ \textbf{val} \ x \ = \ s_0; \ s \ : \ \tau}\ [\textsc{Val}]$$

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

$$\frac{\Gamma \mid \Delta \vdash \ b_0 \ : \ (\overline{\tau}, \ \overline{\sigma}) \to \tau_0 \qquad \overline{\Gamma \vdash \ v \ : \ \tau} \qquad \overline{\Gamma \mid \Delta \vdash \ b \ : \ \sigma}}{\Gamma \mid \Delta \vdash \ b_0(\overline{v}, \ \overline{b}) \ : \ \tau_0}\ [\textsc{App}]$$

$$\frac{\Gamma \mid \Delta \vdash \ b \ : \ \textbf{Cap} \ \tau_1 \ \tau_2 \qquad \Gamma \vdash \ v \ : \ \tau_1}{\Gamma \mid \Delta \vdash \ \textbf{do} \ b(v) \ : \ \tau_2}\ [\textsc{Do}]$$

$$\frac{\Gamma \mid \Delta, \ c : \ \textbf{Cap} \ \tau_1 \ \tau_2 \vdash \ s_0 \ : \ \tau \qquad \Gamma, \ x : \ \tau_1 \mid \Delta, \ k : \ \textbf{Cap} \ \tau_2 \ \tau \vdash \ s \ : \ \tau}{\Gamma \mid \Delta \vdash \ \textbf{try} \ \{ \ (c) \Rightarrow s_0 \ \} \ \textbf{with} \ \{ \ (x, \ k) \Rightarrow s \ \} \ : \ \tau}\ [\textsc{Try}]$$

Figure 5 Type system of System Ξ.

which treats capabilities and function blocks differently. Capabilities cannot have block parameters as this would allow blocks to leave their defining scopes and therefore break the second-class property. Finally, handling effects is done in explicit capability-passing style. In **try** $\{ \ (c) \Rightarrow s_0 \ \}$ **with** $\{ \ (x, \ k) \Rightarrow s \ \}$ the capability is bound to c in the scope of the handled statement s_0. The implementation of the capability binds its value parameter x and the continuation k in the implementation statement s.

Typing rules Figure 5 defines the typing rules for System Ξ. Typing for values is entirely standard. Note that values as well as statements are typed against value types. In contrast, blocks are typed against block types, that is, they have either function type or capability type. Moreover, there are two kinds of environments in the typing judgment of blocks, namely Γ for value bindings and Δ for block bindings. The same is true for statement typing. In particular, when typing a function block, the value parameters are added to the value environment and the block parameters are added to the block environment.

Apart from distinguishing two kinds of environment, the rules for sequencing (Val), returning (Ret), block definition (Def) and function block calls (App) are standard. Compared to the original version of System Ξ we have an additional rule (Do) for performing capabilities, a consequence of syntactically distinguishing them from function blocks as mentioned above. A capability has type **Cap** τ_1 τ_2 where τ_1 is the type of its parameter and τ_2 is the return type. Otherwise the rule is essentially the same as the one for calling function blocks. The crucial rule is Try. The handler makes the capability available in the scope of the handled statement by adding a binding for it to the block environment for the handled statement. Since blocks cannot be returned, the capability can only be used in that scope, thus guaranteeing effect safety without any visible effect system. In the implementation statement, the continuation parameter has capability type. This is in contrast with the original version of System Ξ where it has function type. The reason for treating continuations as capabilities is to simplify the translation to our generalized version of Λ_{cap}. For System Ξ this does not add expressivity as capabilities can be used in the same contexts as function blocks.

3.1.2 Target Calculus Λ_{cap}

The target calculus Λ_{cap} also is a calculus with lexical effect handlers in explicit capability-passing style. In contrast to System Ξ, it features first-class functions and has explicit regions and subregion evidence.

Syntax The syntax is given in Figure 6. It is again in fine-grain call-by-value, but does not distinguish blocks from values. So blocks are now values, making them first class. This is also reflected on the level of types in that function and capability types are not in a separate syntactic category. But, to ensure effect safety, there are regions and subregion evidence. Regions are either region variables or the toplevel region. Evidence is a witness of the subregion relationship between regions. It is either an evidence variable, the trivial evidence $\mathbb{0}$, or the composition $e \oplus e$ of evidence.

Functions do not distinguish value and block parameters but can now additionally abstract over a list of regions and a list of evidence. Accordingly, when calling a function, corresponding lists of regions and evidence have to be supplied. Furthermore, each function is annotated with the region it is supposed to run in. When calling a capability, it needs to be supplied with evidence but not with a region. An effect handler not only abstracts a capability but additionally a region and an evidence variable for the handled statement, whereas the implementation statement stays the same as in System Ξ. The constructs for sequencing and return are also the same. There is no construct for the definition of a local function, as it can be easily defined as syntactic sugar using sequencing now that functions are first class.

Typing rules Figure 7 shows the typing rules for Λ_{cap}. In contrast to System Ξ, there is only one typing environment, however, in addition to value bindings it can also contain regions and evidence bindings. Moreover, the typing judgment for statements has as an additional component the region in which the statement is typed.

As functions can abstract region and evidence parameters, these are added to the environment when typing the function body (rule Fun). Moreover, the function has to run in the region its body is typed in. This is also visible in the type of the function. When sequencing two statements, both are typed in the same region as the compound statement. Returning a result can be typed in any region. When applying a function (rule App), its region arguments are substituted not only in the return type, but also in the types for the evidence and value arguments when typing them. Furthermore, when substituted in the region annotated in the function type, the resulting region has to coincide with the region the applied function is typed in. This allows functions to be region-polymorphic.

Rule Do defines the typing for performing capabilities. A capability can only be called in a subregion of the region annotated in its type. This is witnessed by the evidence supplied in the call. The region annotated in the type of the capability c abstracted at the handled

Syntax of Terms:

Statements			
s	$::=$	**val** $x \;=\; s;\; s$	sequencing
	$\vert$	**return** v	returning values
	$\vert$	$v[\overline{\rho}\;;\;\overline{e}](\overline{v})$	calling functions
	$\vert$	**do** $v[e](v)$	performing capabilities
	$\vert$	**try** $\{\;[r\;;\;n](c) \Rightarrow s\;\}$ **with** $\{\;(x,\;k) \Rightarrow s\;\}$	handling effects
Values			
v	$::=$	$x,\; f,\; k,\; c$	variables
	$\vert$	$()$ $\vert$ 0 $\vert$ 1 $\vert$ … $\vert$ true $\vert$ …	constants
	$\vert$	$\{\;[\overline{r}\;;\;\overline{n:\;\gamma}](\overline{x:\;\tau})$ **at** $\rho \Rightarrow s\}$	closures
Evidence			
e	$::=$	$n,\;\ldots$	evidence variables
	$\vert$	$\mathbb{0}$	reflexive evidence
	$\vert$	$e\;\oplus\;e$	transitive evidence

Syntax of Types:

Types			
τ	$::=$	Unit $\vert$ Int $\vert$ Bool $\vert$ …	primitives
	$\vert$	$\forall[\overline{r}\;;\;\overline{\gamma}](\overline{\tau}) \to\!\rho\;\tau$	functions
	$\vert$	**Cap** $\rho\;\tau\;\tau$	capabilities
Regions			
ρ	$::=$	r	region variable
	$\vert$	$\top$	toplevel region
Constraints			
γ	$::=$	$\rho\;\sqsubseteq\;\rho$	subregion

Environments:

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

Names:

Variables $x,\; y,\; f,\; g,\; k,\; c\;\in$ x, y, f, g, k Fail, Choice, …

Syntactic Sugar:

def $f \;=\; v \;\doteq\;$ **val** $f \;=\;$ **return** v

Figure 6 Syntax of Λ_{cap}.

statement s_0 of an effect handler (rule Try) is the fresh region r also abstracted there. Region r is also the region which s_0 is typed in. Thus, c can only be called in a subregion of r, *i.e.*, within the handled statement. This ensures effect safety. The additionally abstracted evidence n witnesses that r is a subregion of the region ρ the overall statement is typed in. The regions and subregion evidence hence precisely reflect how handlers are nested. The implementation statement is typed in the outer region ρ which is also the region for the continuation. In constrast to the original version of Λ_{cap}, the continuation has capability type, not function type. This allows to call the continuation not only in region ρ but also in

Value Typing

$$\boxed{\begin{array}{ccc} \Gamma \vdash & v & : \tau \\ \uparrow & \uparrow & \downarrow \end{array}}$$

$$\frac{\Gamma(x) \;=\; \tau}{\Gamma \vdash\; x \;:\; \tau}\;[\textsc{Var}] \qquad \frac{}{\Gamma \vdash\; 3 \;:\; \mathsf{Int}}\;[\textsc{Lit}]$$

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

Statement Typing

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

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

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

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

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

Evidence Typing

$$\boxed{\begin{array}{ccc} \Gamma \vdash & e & : \gamma \\ \uparrow & \uparrow & \downarrow \end{array}}$$

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

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

Figure 7 Type system of Λ_{cap}.

any subregion of ρ. Our version is thus more expressive as continuations can, for example, be called under further effect handlers, *i.e.*, we lift the restriction of scoped continuations imposed by the original version. This is important in order to fully support a lift-inference translation from System Ξ since there no such restriction for continuations exists.

The typing of evidence is rather straightforward. Evidence variables are looked up in the environment. The trivial evidence $\mathbb{0}$ witnesses that any region is a subregion of itself. Composition of evidence shows that subregioning is transitive.

3.2 Operational Semantics

We define the operational semantics of the two calculi in terms of an abstract machine. The abstract machine for Λ_{cap} is essentially the same as the one for System Ξ. There are only two minor differences. First, there is no stepping rule for the definition of local functions in Λ_{cap} as there is no separate construct for them. Second, there are regions and evidence. However, the latter are irrelevant for the machine semantics as it proceeds by searching delimiters with labels on the meta stack and does not use region and evidence information.

As the operational semantics of both languages is so similar, it is not that interesting with regards to lift inference. Nevertheless, knowing the operational semantics helps to understand lifting, in particular with respect to continuations. Therefore, we briefly sketch how the machine works, and also where it differs from the original versions of the two calculi. A detailed discussion is given in Appendix B.1.

A machine state $\langle\, s \,\|\, \mathsf{K} \,\rangle$ contains a statement s to be evaluated and a runtime meta stack K which is a list of delimited stacks. A stack is a list of frames ending with a delimiter $\#_l$ containing a label l. This is a minor technical difference to the original version of the machine which treats delimiters as regular frames and does not explicitly segment the meta stack into delimited stacks. It facilitates the correctness proof of the CPS translation from our generalized version of Λ_{cap} to System F.

The machine implements multi-prompt delimited control [9]. Upon execution of a handler statement, a fresh label is generated and a new stack consisting only of a delimiter with that label is pushed onto the meta stack.

$$\textit{(try)} \qquad \langle\, \textbf{try}\ \{\ (c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \} \,\|\, \mathsf{K} \,\rangle \ \ \rightarrow\ \ \langle\, s_0[c \mapsto v] \,\|\, \#_l :: \ \mathsf{K} \,\rangle$$
$$\text{where } l\ =\ \texttt{generateFresh}() \text{ and } v\ =\ \textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}$$

Execution then continues with the handled statement where the abstracted capability variable is replaced by a runtime capability which contains the just generated label l and the handler implementation. When encountering a call to a capability the machine transitions to unwinding mode and pops stacks off the meta stack until the correct label is found. These stacks are collected in a resumption which is used as continuation.

It is exactly this runtime search of correct handlers that we seek to avoid by using lifting information. To make this possible, we make sure that the evidence passed to capabilities precisely reflects the labels on the runtime meta stack and we take care that this invariant is preserved during the execution of the machine. In fact, at runtime each evidence becomes a list of appropriate labels which is adapted as the machine proceeds. The invariant ensures that evidence always contains the correct lifting information. While being irrelevant for the machine semantics, this is critically important for the CPS translation, since the latter uses evidence instead of labels (see Subsection 3.3).

The main difference to the original version of the machine is how a call of a continuation proceeds. The reason we have to treat this differently is that, compared to the original version of Λ_{cap}, we allow the use of continuations under further handlers, in order to fully support System Ξ in the lift-inference translation. A continuation may contain capabilities which have been provided with evidence. When calling the continuation under further handlers in the implementation statement, additional delimiters are installed on the meta stack that were not present when the continuation was created. Thus, the evidence for the capabilities inside the continuation does not precisely reflect the labels on the meta stack which violates the critical invariant described above.

To make the evidence correct again, we have to make the additional delimiters "invisible" for the continuation. This is achieved by treating continuations as capabilities as well. This way, when a continuation is called, it first captures the additional stacks with these delimiters by unwinding as described above (which is again reflected by the evidence passed to the continuation itself), and packages them into one resumption frame. Only then, the continuation is executed in the usual way by rewinding. Such a resumption frame acts a bit like an "underflow" frame [10] when returning to it, in the sense that execution then first continues with that resumption. When unwinding, however, it is treated just as another ordinary frame so that the stacks inside of it do not inferfere with the unwinding. Hence,

Translation of Statements:

$$\begin{array}{lcl} & \vdots & \\ \mathcal{S}[\![\ \textbf{do}\ v_0[e](v)\ ]\!] & = & \mathcal{E}[\![\ e\ ]\!] \quad \mathcal{T}[\![\ \tau_2\ ]\!] \quad (\mathcal{V}[\![\ v_0\ ]\!]\ \mathcal{V}[\![\ v\ ]\!]) \\ \mathcal{S}[\![\ \textbf{do}\ k[e](v)\ ]\!] & = & \mathcal{E}[\![\ e\ ]\!] \quad \mathcal{T}[\![\ \tau_2\ ]\!] \quad (\lambda k_0.\ \mathcal{V}[\![\ k\ ]\!]\ \mathcal{V}[\![\ v\ ]\!]\ k_0) \\ \mathcal{S}[\![\ \textbf{try}\ \{\ [r;\ n](c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}\ ]\!] & = & \textsc{Reset}\ ((\Lambda r.\ \lambda n.\ \lambda c.\ \mathcal{S}[\![\ s_0\ ]\!]) \\ & & \qquad (\textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \mathcal{T}[\![\ \tau\ ]\!]) \quad (\textsc{Lift}) \quad (\lambda x.\ \lambda k.\ \mathcal{S}[\![\ s\ ]\!])) \end{array}$$

Auxiliary Definitions:

$$\begin{array}{lcl} \textsc{Cps}\ R\ A & = & (A \to R) \to R \end{array}$$

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

Figure 8 CPS Translation from Λ_{cap} to System F.

when a call to a capability inside the continuation is encountered, it only sees the labels present on the meta stack when the continuation was created so that its evidence is correct.

This difference in how continuations are treated compared to the original version does not impact the final result of the execution for all programs that can be written in the original versions of the calculi. It leads, however, to another minor difference to the original machine. As the continuation capabilities are always delimited by the next label on the meta stack at the point of their creation, execution of a closed statement s always starts with a delimiter with a special toplevel label on the otherwise empty meta stack, that is, in state $\langle\ s\ \|\ \#_{\mathsf{start}} :: \bullet\ \rangle$. This ensures that there also is a delimiting label for continuations of **try**-statements in the toplevel region.

3.3 CPS Translation to System F

For the original version of Λ_{cap} Schuster et al. [36] give a typability- and semantics-preserving CPS translation to pure System F. This CPS translation carries over almost unchanged to our version of Λ_{cap}. Still, it is instructive to briefly repeat the core idea, in particular, to see how evidence enables efficient compilation.

The idea of the CPS translation is to use evidence information to decide how to lift a capability, *i.e.*, which parts of the runtime meta stack need to be captured. Or put in terms of the operational semantics, which delimiters have to be jumped over when unwinding. As a result, no runtime search for the correct label on the meta stack is needed anymore. To this end, the translation targets so-called iterated CPS [34], which uses one continuation parameter for each stack delimited by a label.

The full CPS translation is given in Appendix B.2. Figure 8 only shows how handlers and calls of capabilities are translated. Note that the translation is actually defined over typing derivations, but we only write the term here. In the translation of a handler the handled statement becomes a function applied to three arguments. The region parameter r represents the polymorphic answer type that has to be instantiated appropriately (τ and ρ are the overall type and region of the **try**-statement as in typing rule Try), the evidence variable n is instantiated with the function Lift and the capability parameter c is instantiated with the translated implementation statement. The whole term is then applied to an empty continuation acting as a delimiter by meta function Reset. The function Lift increases the number of continuation parameters of its argument m by one, hence m is lifted to a different region by capturing one more delimited stack. Note that the explicitly abstracted type parameter a is the immediate return type (see also the application of a capability below), while the (answer) types R, R' are determined by the surrounding regions.

In the CPS translation of performing capabilities the capability is applied to its argument. The resulting term is then fed into the translated evidence of the capability (in the type argument $\mathcal{T}[\![\ \tau_2\ ]\!]$, τ_2 is the return type of the capability as in typing rule Do). This evidence always eventually consists of a composition of evidence variables bound at handlers[6], which means that it is a composition of Lift-terms. Thus, the translated evidence term determines how far the capability is lifted, that is, how many stacks of the meta stack are captured.

As explained above we also treat the continuation as a capability and provide it with evidence. This ensures that the handler capabilities inside the continuation are correctly lifted since the continuation itself is lifted to the correct region. The CPS translation for continuation capabilities is almost the same as for handler capabilities, the only difference being that the applied continuation is η-expanded. This is necessary in order to make the following simulation theorem true.

▶ **Theorem 1** (Simulation for the CPS Translation).
If $\vdash\ \mathsf{M}\ \mathit{ok}$ *and* $\mathsf{M} \to \mathsf{M}'$, *then* $\mathcal{M}[\![\ \mathsf{M}\ ]\!] \to^* \mathcal{M}[\![\ \mathsf{M}'\ ]\!]$.

M denotes a machine state. The operational semantics of Λ_{cap} hence corresponds to reduction in System F. As a corollary we obtain that evaluation is preserved by the CPS translation.

▶ **Corollary 2** (Evaluation for the CPS Translation).
If $\emptyset \mid \top \vdash\ s\ :\ \mathit{Int}$ *and* $\langle\ s \parallel \#_{\mathit{start}} :: \bullet\ \rangle \ \to^*\ \langle\ \textbf{\textit{return}}\ v \parallel \bullet\ \rangle$,
then $\textsc{Reset}(\mathcal{S}[\![\ s\ ]\!])\ \mathtt{done}\ \to^*\ \mathtt{done}\ \mathcal{V}[\![\ v\ ]\!]$.

Here `done` is a special toplevel continuation and the Reset is necessary due to the presence of the toplevel label. As the operational semantics of our version of Λ_{cap} differs a bit from the original version, the proof of the simulation theorem and the necessary translation of the runtime constructs had to be adapted. This is shown in Appendix B.3. In contrast, the proof of the following typability preservation stays almost unchanged.

▶ **Theorem 3** (Typability Preservation for the CPS Translation).
If $\Gamma \mid \rho \vdash\ s\ :\ \tau$, *then* $\mathcal{T}[\![\ \Gamma\ ]\!] \vdash\ \mathcal{S}[\![\ s\ ]\!]\ :\ \textsc{Cps}\ \mathcal{T}[\![\ \rho\ ]\!]\ \mathcal{T}[\![\ \tau\ ]\!]$.
If $\Gamma \vdash\ v\ :\ \tau$, *then* $\mathcal{T}[\![\ \Gamma\ ]\!] \vdash\ \mathcal{V}[\![\ v\ ]\!]\ :\ \mathcal{T}[\![\ \tau\ ]\!]$.

3.4 Lift-Inference Translation

We now present the lift-inference translation from System Ξ to Λ_{cap}. This is our main contribution. The translation is defined over typing derivations of System Ξ and is supposed to take well-typed terms in System Ξ to well-typed terms in Λ_{cap}, so we translate types and terms. In the clauses for the terms we only write the term instead of the whole typing derivation.

The translation is defined in Figure 9. As the two calculi are quite similar, the translation mainly proceeds by endowing terms in System Ξ with appropriate region and evidence abstractions and applications in the right places. To this end, we maintain a lifting environment E during the translation which is used to remember which blocks have been bound so far and in what region, while we descend recursively.

This environment is modeled as a record consisting of four components. First, it contains the current region ρ of the term to be translated. Second, it contains the block environment

[6] It may further be interspersed with trivial evidence which is, however, translated to the identity function.

Region-and-Evidence Environment:

$$\begin{array}{lcl}
\mathsf{E} & = & \{\ \rho,\ \Delta,\ m:\ \mathsf{Map}(\mathsf{dom}(\Delta),\ \mathsf{Reg}\ \times \mathsf{Ev}),\ \Gamma_E\ \}
\end{array}$$

Translation of Types:

$$\begin{array}{lcl}
\mathcal{T}\llbracket\ (\tau) \to \tau_0\ \rrbracket_\rho & = & \forall[r;\ r\ \sqsubseteq\ \rho]\ (\tau) \to r\ \tau_0 \\
\quad \text{where } r\ =\ \texttt{generateFresh}() & & \\
\mathcal{T}\llbracket\ (\sigma) \to \tau_0\ \rrbracket_\rho & = & \forall[r,\ r_f;\ r\ \sqsubseteq\ \rho,\ r\ \sqsubseteq\ r_f]\ (\mathcal{T}\llbracket\ \sigma\ \rrbracket_{r_f}) \to r\ \tau_0 \\
\quad \text{where } r,\ r_f\ =\ \texttt{generateFresh}() & & \\
\mathcal{T}\llbracket\ \textbf{Cap}\ \tau_1\ \tau_2\ \rrbracket_\rho & = & \textbf{Cap}\ \rho\ \tau_1\ \tau_2
\end{array}$$

Translation of Environments:

$$\begin{array}{lcl}
\mathcal{T}\llbracket\ \emptyset\ \rrbracket^{\mathsf{E}} & = & \emptyset \\
\mathcal{T}\llbracket\ \Delta,\ f:\ \sigma\ \rrbracket^{\mathsf{E}} & = & \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}},\ f\ :\ \mathcal{T}\llbracket\ \sigma\ \rrbracket_{\mathcal{R}\llbracket\ f\ \rrbracket^{\mathsf{E}}}
\end{array}$$

Translation of Blocks:

$$\begin{array}{lcl}
B\llbracket\ f\ \rrbracket^{\mathsf{E}} & = & f \\
B\llbracket\ \{\ (x:\ \tau) \Rightarrow s_0\ \}\rrbracket^{\mathsf{E}} & = & \{\ [r;\ n:\ r\ \sqsubseteq\ \mathsf{E}.\rho](x:\ \tau)\ \textbf{at}\ r \Rightarrow \mathcal{S}\llbracket\ s_0\ \rrbracket^{\mathsf{E}\ \oplus\ n}\ \} \\
\quad \text{where } r,\ n\ =\ \texttt{generateFresh}() & & \\
B\llbracket\ \{\ (f:\ \sigma) \Rightarrow s_0\ \}\rrbracket^{\mathsf{E}} & = & \\
\multicolumn{3}{l}{\quad \{\ [r,\ r_f;\ n:\ r\ \sqsubseteq\ \mathsf{E}.\rho,\ n_f:\ r\ \sqsubseteq\ r_f](f:\ \mathcal{T}\llbracket\ \sigma\ \rrbracket_{r_f})\ \textbf{at}\ r \Rightarrow \mathcal{S}\llbracket\ s_0\ \rrbracket^{\mathsf{E}'}\ \}} \\
\multicolumn{3}{l}{\qquad \text{where } r,\ n,\ r_f,\ n_f\ =\ \texttt{generateFresh}() \text{ and } \mathsf{E}'\ =\ \mathsf{E}\ \oplus n,\ f \mapsto (r_f\ \mid n_f),\ r_f,\ n_f:\ r\ \sqsubseteq\ r_f}
\end{array}$$

Translation of Statements:

$$\begin{array}{lcl}
\mathcal{S}\llbracket\ \textbf{return}\ v\ \rrbracket^{\mathsf{E}} & = & \textbf{return}\ v \\
\mathcal{S}\llbracket\ \textbf{val}\ x\ =\ s_0;\ s\ \rrbracket^{\mathsf{E}} & = & \textbf{val}\ x\ =\ \mathcal{S}\llbracket\ s_0\ \rrbracket^{\mathsf{E}};\ \mathcal{S}\llbracket\ s\ \rrbracket^{\mathsf{E}} \\
\mathcal{S}\llbracket\ \textbf{def}\ f\ =\ b;\ s\ \rrbracket^{\mathsf{E}} & = & \textbf{def}\ f\ =\ B\llbracket\ b\ \rrbracket^{\mathsf{E}};\ \mathcal{S}\llbracket\ s\ \rrbracket^{\mathsf{E},\ f\ \mapsto\ (\mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}\ \mid\ \mathcal{C}\llbracket\ b\ \rrbracket^{\mathsf{E}})} \\
\mathcal{S}\llbracket\ b(v)\ \rrbracket^{\mathsf{E}} & = & B\llbracket\ b\ \rrbracket^{\mathsf{E}}[\mathsf{E}.\rho;\ \mathcal{C}\llbracket\ b\ \rrbracket^{\mathsf{E}}\ ](v) \\
\mathcal{S}\llbracket\ b(b_0)\ \rrbracket^{\mathsf{E}} & = & B\llbracket\ b\ \rrbracket^{\mathsf{E}}[\mathsf{E}.\rho,\ \mathcal{R}\llbracket\ b_0\ \rrbracket^{\mathsf{E}};\ \mathcal{C}\llbracket\ b\ \rrbracket^{\mathsf{E}},\ \mathcal{C}\llbracket\ b_0\ \rrbracket^{\mathsf{E}}\ ](B\llbracket\ b_0\ \rrbracket^{\mathsf{E}}) \\
\mathcal{S}\llbracket\ \textbf{do}\ c(v)\ \rrbracket^{\mathsf{E}} & = & \textbf{do}\ c[\ \mathcal{C}\llbracket\ c\ \rrbracket^{\mathsf{E}}\ ](v) \\
\mathcal{S}\llbracket\ \textbf{try}\ \{\ (c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}\ \rrbracket^{\mathsf{E}} & = & \\
\multicolumn{3}{l}{\quad \textbf{try}\ \{\ [r\ ;\ n:\ r\ \sqsubseteq\ \mathsf{E}.\rho](c) \Rightarrow \mathcal{S}\llbracket\ s_0\ \rrbracket^{\mathsf{E}\ \oplus n,\ c\ \mapsto\ (r\ \mid\ \mathbb{0})}\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow \mathcal{S}\llbracket\ s\ \rrbracket^{\mathsf{E},\ k\ \mapsto\ (\mathsf{E}.\rho\ \mid\ \mathbb{0})}\ \}} \\
\multicolumn{3}{l}{\qquad \text{where } r,\ n\ =\ \texttt{generateFresh}()}
\end{array}$$

Lookup and Adaptions of Region-and-Evidence Environment:

$$\begin{array}{lcl}
\mathcal{R}\llbracket\ f\ \rrbracket^{\mathsf{E}} & = & \rho \qquad \text{where } (\rho,\ e)\ =\ \mathsf{E}.m(f) \\
\mathcal{R}\llbracket\ w\ \rrbracket^{\mathsf{E}} & = & \mathsf{E}.\rho \\
\mathcal{C}\llbracket\ f\ \rrbracket^{\mathsf{E}} & = & e \qquad \text{where } (\rho,\ e)\ =\ \mathsf{E}.m(f) \\
\mathcal{C}\llbracket\ w\ \rrbracket^{\mathsf{E}} & = & \mathbb{0} \\
 & & \\
\emptyset\ \oplus n & = & \emptyset \\
(m,\ f \mapsto (\rho\ \mid n_0))\ \oplus n & = & m\ \oplus n,\ f \mapsto (\rho\ \mid n\ \oplus n_0) \\
 & & \\
\{\ \rho,\ \Delta,\ m,\ \Gamma_E\ \}\ \oplus n & = & \{\ r,\ \Delta,\ m\ \oplus n,\ (\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho)\ \} \quad \text{where } n:\ r\ \sqsubseteq\ \rho \\
\{\ \rho,\ \Delta,\ m,\ \Gamma_E\ \},\ f \mapsto (\rho'\ \mid e) & = & \{\ \rho,\ (\Delta, f),\ (m,\ f \mapsto (\rho'\ \mid e)),\ \Gamma_E\ \} \\
\{\ \rho,\ \Delta,\ m,\ \Gamma_E\ \},\ r,\ n:\ \rho\ \sqsubseteq\ r & = & \{\ \rho,\ \Delta,\ m,\ (\Gamma_E,\ r,\ n:\ \rho\ \sqsubseteq\ r)\ \}
\end{array}$$

Figure 9 Lift-Inference Translation from System Ξ to Λ_{cap}.

Δ of the term to be translated. The types of the blocks in Δ are not needed in the lifting environment and we usually omit them, but it eases presentation a bit to just write Δ. The third component is a map m from the domain $\mathsf{dom}(\Delta)$ of the block environment to pairs of regions (Reg) and evidence (Ev). The region stands for the region of the definition-site of the entry and the evidence is supposed to witness that the current region ρ is a subregion of the definition-site region. This invariant is enabled by the second-class property of blocks which guarantees that each call-site of a block is in a subregion of the region of the definition-site. To maintain the invariant, the evidence for each block in the map has to be adapted when the current region changes during the translation. As a fourth component the lifting environment contains a typing environment Γ_E which consists of all the regions and evidence that are present in map m. The above invariant can now more formally be captured in the following definition.

▶ **Definition 4** (Soundness of Region-and-Evidence Environment).
We call $\mathsf{E} = \{\ \rho,\ \Delta,\ m,\ \Gamma_E\ \}$ *sound if* $\Gamma_E \vdash \mathcal{C}[\![\ f\ ]\!]^{\mathsf{E}} : \rho \sqsubseteq \mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}}$ *for all* $f \in \mathit{dom}(\Delta)$.

Here the functions $\mathcal{R}[\![\ \cdot\ ]\!]$ and $\mathcal{C}[\![\ \cdot\ ]\!]$ are lookup functions for the region and evidence component of blocks in the map m, respectively (see Figure 9).

Translation of values Values in System Ξ are either variables or constants, both of which are translated trivially to the same terms in Λ_{cap}. Similarly, value types τ in System Ξ are only base types and thus remain unchanged. Accordingly, value environments Γ need not be translated. Therefore, these parts of the calculus are omitted from the presentation in Figure 9.

Translation of blocks Blocks in System Ξ are translated to values in Λ_{cap}. Just as value variables, block variables are translated trivially. For function blocks we only show the case of a function with one parameter and treat the cases for a value parameter and a block parameter separately to ease presentation. Multi-arity functions are translated accordingly in the obvious way.

In either case the translated function abstracts a fresh region r and fresh evidence n which witnesses that r is a subregion of the current region. Moreover, the function is annotated to run in the abstracted region r, *i.e.*, it is region-polymorphic, but the evidence enforces the constraint that the actual region in which the function will run must be a subregion of the current region of the definition-site.

In the case of a value parameter the body of the function is translated recursively but the lifting environment E is adapted to $\mathsf{E} \oplus n$. This has three effects. The current region is changed to be the abstracted region r. The evidence component of all entries in the map m of E is composed with the additional evidence n. This is necessary to maintain the above-mentioned soundness invariant since the current region has changed. Moreover, r and n are added to the typing environment Γ_E.

In the case of a block parameter $f : \sigma$ the translated function abstracts an additional region r_f which stands for the region of the definition-site of f and an additional evidence parameter $n_f : r \sqsubseteq r_f$ witnessing that the region r the function will run in is a subregion of r_f. As the definition-site region of f is only known when it is actually instantiated, it is necessary to abstract over it. This region is also used to translate the type σ of f. The constraint that n_f imposes says that the block the parameter f later is instantiated with must be defined in a superregion of the region in which the whole function is called. The lifting environment for the translation of the body is first adapted in the same way as in the case of a value parameter, but must then be further adapted by adding an entry for f. This entry consists of the pair $(r_f \mid n_f)$ which satifies the soundness invariant since the current region now is r. Moreover, r_f and n_f must be added to the typing environment Γ_E.

Note that extending the lifting environment E with new entries for blocks and with additional region and evidence variables is both written as comma-separated concatenation, but that the two extensions affect different components of E.

Translation of block types The translation of block types does not need the lifting environment as additional input but only a region standing for the region of the definition-site of the block. For functions, the additionally abstracted region and evidence parameters for the translated function itself and each of its block parameters are directly reflected in the type. For capability types the given region is simply added as the region for the capability. The translation of block environments proceeds by pointwise translation of the types of the block bindings. However, as the region input must be the definition-site region for each block, we have to look this region up in the lifting environment E using the lookup function $\mathcal{R}[\![\ \cdot\ ]\!]$ for regions. The translation of block environments therefore does need E as input.

Translation of statements The translation for returning values is trivial and for sequencing of statements we simply translate the substatements recursively with the same environment. The definition of a local block b is a bit more interesting. It is translated to the definition of the translated block (note that this is syntactic sugar in Λ_{cap}), but for the translation of the remaining statement we have to adapt the lifting environment E by inserting an entry for this newly defined function. Now there are two cases for b. Either it is a block variable g (*i.e.*, the definition is just aliasing), then it must be in the environment and we have to look up the correct region and evidence for g in E. Or it is a block value w, then there is no binding in the environment yet. In this case, the region of the definition-site of the block is the current region and hence the correct evidence is $\mathbb{0}$. The translation for this case could thus instead be defined as

$$\mathcal{S}[\![\ \textbf{def}\ f\ =\ w;\ s\ ]\!]^{\mathsf{E}} \quad = \quad \textbf{def}\ f\ =\ B[\![\ w\ ]\!]^{\mathsf{E}};\ \mathcal{S}[\![\ s\ ]\!]^{\mathsf{E}\ +\ f\ \mapsto\ (\mathsf{E}.\rho\ \mid\ \mathbb{0})}$$

However, since the lookup functions $\mathcal{R}[\![\ \cdot\ ]\!]$ and $\mathcal{C}[\![\ \cdot\ ]\!]$ are defined to yield exactly the above results for block values, we do not need to distinguish cases in the translation in Figure 9.

Calling a function block b is translated to calling the translated block. We again only show the cases for one value argument and one block argument, but the generalization to multi-arity functions is again done in the obvious way. In either case the translated block has abstracted a region and an evidence parameter. The region parameter stands for the region in which the function runs, so we instantiate it with the current region. Remember that the evidence parameter must witness that the region parameter is a subregion of the definition-site region of b. But as the region parameter was instantiated with the current region we can exploit the carefully maintained soundness and simply look the correct evidence up in the lifting environment. Note that if b is an anonymous block value, the definition-site region is the current region and so the trivial evidence yielded by the lookup function for evidence is correct. In the case of a block parameter we have to translate the block argument b_0 as well, of course. But we additionally have to supply a region and evidence for the corresponding parameters that have been abstracted for the block parameter. Both of these can simply be looked up, too, since the region lookup will yield the region of the definition-site of b_0 and soundness again makes sure that the corresponding evidence is correct.

Translating an applied capability is similar to calling a function with a value parameter, the needed evidence is simply looked up. The difference is just that no region needs to be supplied as capabilities are not region-polymorphic. The region of their definition-site always is the fresh region abstracted at the translation of the corresponding handler statement. This can be seen in the translation of a handler which consists of the translation of the handler statement and the translation of the implementation statement. The former is similar to how function blocks with a block parameter are translated. The lifting environment E is first adapted with the newly abstracted evidence n and then an entry for the capability c is added.

As the region for this entry now is the current one, the corresponding evidence is trivial, *i.e.*, $\mathbb{0}$. Note that since the fresh abstracted region is already added to the typing environment Γ_E by $\mathsf{E} \oplus n$, it is not necessary to do this in an extra step. For the implementation statement, no region and evidence is abstracted, so we only have to add an entry for the continuation parameter k to E. The region for k is the current one, that is, the one the whole handler is defined in. The evidence for k thus again is $\mathbb{0}$.

3.4.1 Example

To illustrate the lift-inference translation, we consider again the example from Subsection 2.2, or more precisely the definition of call.

$$\begin{array}{l}
\textbf{def}\ \mathsf{call}\ =\ \{\ (\mathsf{f}:\ \mathsf{Int} \to \mathsf{Int}) \Rightarrow \\
\quad \textbf{val}\ \mathsf{x}\ =\ \mathsf{f}(1); \\
\quad \textbf{try}\ \{\ (\mathsf{yield}_2) \Rightarrow \mathsf{f}(\mathsf{x})\ \} \\
\quad \textbf{with}\ \{\ (\mathsf{j},\ \mathsf{k}) \Rightarrow 42\ \} \\
\};
\end{array}$$

Starting with the empty lifting environment $\mathsf{E}\ =\ \{\ \top,\ \emptyset,\ \emptyset,\ \emptyset\ \}$, we show how the environment changes as the translation proceeds. In the first step, regions and evidence for call and f are abstracted, the type of f is translated with the abstracted region and the region annotation is added. Thus, we obtain

$$\begin{array}{l}
\textbf{def}\ \mathsf{call}\ =\ \{\ [\mathsf{r_c},\ \mathsf{r_f};\ \mathsf{n_c}:\ \mathsf{r_c} \sqsubseteq \top,\ \mathsf{n_f}:\ \mathsf{r_c} \sqsubseteq \mathsf{r_f}](\\
\quad\quad \mathsf{f}:\ \forall[\mathsf{r};\ \mathsf{r} \sqsubseteq \mathsf{r_f}](\mathsf{Int}) \to \mathsf{r}\ \mathsf{Int} \\
\quad)\ \textbf{at}\ \mathsf{r_c} \Rightarrow \mathcal{S}[\![\ \ldots\ ]\!]^{\mathsf{E}'} \\
\};
\end{array}$$

The lifting environment for the recursive translation of the body is adapted since we have entered a new region and since we have to add a new entry for the block parameter. It becomes

$$\begin{array}{rl}
\mathsf{E}' = & \mathsf{E}\ \oplus \mathsf{n_c},\ \mathsf{f} \mapsto (\mathsf{r_f}\ \mid \mathsf{n_f}),\ \mathsf{r_f},\ \mathsf{n_f}\ :\ \mathsf{r_c} \sqsubseteq \mathsf{r_f} \\
= & \{\ \mathsf{r_c},\ (\mathsf{f}),\ (\mathsf{f} \mapsto (\mathsf{r_f}\ \mid \mathsf{n_f})),\ (\mathsf{r_c},\ \mathsf{n_c}:\ \mathsf{r_c} \sqsubseteq \top,\ \mathsf{r_f},\ \mathsf{n_f}\ :\ \mathsf{r_c} \sqsubseteq \mathsf{r_f})\ \}
\end{array}$$

In the translation of the body, the translation of the first application of f is a simple matter of looking up the current region and the evidence for f in the environment. The effect handler is endowed with a fresh region and evidence and the substatements are translated recursively. As the implementation statement is just a value, it is translated trivially, and we obtain

$$\begin{array}{l}
\textbf{val}\ \mathsf{x}\ =\ \mathsf{f}[\mathsf{r_c};\ \mathsf{n_f}](1); \\
\textbf{try}\ \{\ [\mathsf{r_2};\ \mathsf{n_2}\ :\ \mathsf{r_2} \sqsubseteq \mathsf{r_c}](\mathsf{yield}_2) \Rightarrow \mathcal{S}[\![\ \mathsf{f}(\mathsf{x})\ ]\!]^{\mathsf{E}''}\ \} \\
\textbf{with}\ \{\ (\mathsf{j},\ \mathsf{k}) \Rightarrow 42\ \}
\end{array}$$

The lifting environment is adapted for the new region we have entered and a new entry for the capability is added. Importantly, the evidence for the existing binding for f is adapted and we find

$$\begin{array}{rl}
\mathsf{E}'' = & \mathsf{E}'\ \oplus \mathsf{n_2},\ \mathsf{yield}_2 \mapsto (\mathsf{r_2}\ \mid \mathbb{0}) \\
= & \{\ \mathsf{r_2},\ (\mathsf{f},\ \mathsf{yield}_2),\ (\mathsf{f} \mapsto (\mathsf{r_f}\ \mid \mathsf{n_2}\ \oplus \mathsf{n_f}),\ \mathsf{yield}_2 \mapsto (\mathsf{r_2}\ \mid \mathbb{0})),\ (\Gamma_{\mathsf{E}'},\ \mathsf{r_2},\ \mathsf{n_2}:\ \mathsf{r_2} \sqsubseteq \mathsf{r_c})\ \}
\end{array}$$

The translation of the second application of f is again a simple matter of looking up the current region and the evidence for f in the environment. As the latter is kept sound during the translation, the evidence is exactly right,

$$\mathsf{f}[\mathsf{r_2};\ \mathsf{n_2}\ \oplus\ \mathsf{n_f}](\mathsf{x})$$

3.5 Properties of Lift Inference

For the lift-inference translation to be sensible it should be typability- and semantics-preserving.

Typability preservation For the proof of typability preservation we make heavy use of the soundness of the lifting environment. To do so, we need the following lemma stating that soundness is maintained during the translation.

▶ **Lemma 5** (Soundness of Environments for the Lift-Inference Translation).
All adaptions of lifting environments made by the lift-inference translation take sound environments to sound environments.

This enables the theorem that the translation takes well-typed terms in System Ξ to well-typed terms in Λ_{cap}. The proof is given in Appendix A.1.

▶ **Theorem 6** (Typability Preservation for the Lift-Inference Translation).
For $\mathsf{E} = \{ \rho, \Delta, m, \Gamma_E \}$ *sound,*
if $\Gamma \mid \Delta \vdash s : \tau$, *then* $\Gamma_E, \Gamma, \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \mid \rho \vdash \mathcal{S}[\![s]\!]^{\mathsf{E}} : \tau$;
if $\Gamma \mid \Delta \vdash b : \sigma$, *then* $\Gamma_E, \Gamma, \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \vdash B[\![b]\!]^{\mathsf{E}} : \mathcal{T}[\![\sigma]\!]_{\mathcal{R}[\![b]\!]^{\mathsf{E}}}$;
if $\Gamma \vdash v : \tau$, *then* $\Gamma_E, \Gamma \vdash v : \tau$.

Since the empty environment $\emptyset = \{ \top, \emptyset, \emptyset, \emptyset \}$ is trivially sound, Theorem 6 implies typability preservation for the translation of closed terms starting with the empty environment.

Semantics preservation For semantics preservation note that the operational semantics of both calculi mainly differs in the presence of regions and evidence in Λ_{cap}. As noted before, these are irrelevant for the operational semantics and can thus be erased. Then the only difference is that there is no rule for reduction of function definitions in Λ_{cap}. It is replaced by two consecutive rules. Hence, we obtain the following result. Some more details are given in Appendix A.2.

▶ **Theorem 7** (Evaluation for the Lift-Inference Translation).
If $\emptyset \mid \emptyset \vdash s : \tau$ *and* $\langle s \mid\mid \#_{\mathsf{start}} :: \bullet \rangle \to^{n + k} \langle \textbf{return}\ v \mid\mid \bullet \rangle$,
then $\langle \mathcal{S}[\![s]\!]^{\emptyset} \mid\mid \#_{\mathsf{start}} :: \bullet \rangle \to^{n + 2k} \langle \textbf{return}\ v \mid\mid \bullet \rangle$,
where k is the number of steps for function definitions and n the number of other steps in System Ξ.

Note that together with the corresponding results of the papers we build on (see the overview in Figure 1), the above theorems guarantee typability and semantics preservation along the whole compilation pipeline down to System F, fully proven. Hence, a well-typed term in Effekt is guaranteed to have the same semantics after translation to System F. In particular, type safety implies effect safety and thus guarantees that no effect goes unhandled.

4 Evaluation

To evaluate our approach, we have implemented lift inference for the Effekt language. This way, we could close the gap illustrated in Section 1 and write benchmark programs directly in Effekt. We first describe the implementation and some limitations, before we discuss the benchmark results.

4.1 Implementation

Effekt is a functional language with support for lexical effect handlers, effect inference, data types, type polymorphism, interface types that generalize functions, backtrackable local state, and many more features. The implementation of the compiler amounts to around 23k lines of code in Scala. For this paper, we have extended the Effekt compiler in two ways. We have implemented the lift-inference translation presented in Subsection 3.4, and we have implemented a new backend targeting SML in continuation-passing style.

4.1.1 Lift Inference

The overview in Figure 1 doubles as an overview over the compiler phases in the Effekt compiler. A translation of the source language to explicit capability-passing System Ξ, which is called *Core* in the implementation, was already implemented by Brachthäuser et al. [4]. To evaluate the feasibility of the translation described in this paper, we have added a new intermediate representation that corresponds to Λ_{cap}, which is called *Lifted* in the implementation. Lifted is typed and includes explicit subregion evidence, but regions are erased from the type level.

Both Core and Lifted differ from the presentation in this paper in that they also support various other features of the language, such as data types, pattern matching, local mutable state, and more. Like Leijen [21], the Effekt compiler also distiguishes potentially effectful expressions from pure expressions, in order to generate more efficient code. The implementation of lift inference itself is a straightforward translation of the algorithm presented in Section 3.4 to Scala. The lifting environment E is implemented as a `Map[Symbol, List[Lift]]` mapping block variables (*i.e.*, `Symbol`s) to a chain of evidence variables (*i.e.*, `Lift`s) that witness the subregion relationship.

4.1.2 SML Backend in CPS

The translation from Λ_{cap} to System F in iterated continuation-passing style is conceptually described by Schuster et al. [36]. However, they do not present an implementation. As a second implementation step for this paper, we have thus implemented a translation from Lifted to Standard ML [26] in continuation-passing style. Specifically, we target MLton since it is a whole program optimizing compiler. We conjectured that MLton could discover many of the static abstractions identified by Schuster et al. [35] at compile-time and thus heavily optimize the generated programs in CPS. We can, however, imagine that a setting with separate compilation could profit from lift inference as well. For example, our approach might be applied within a compilation unit and one might moreover rely on link-time optimizations or just-in-time compilation to obtain further improvements from lift inference at runtime. We leave closer investigation of this to future work.

While the translation conceptually translates effectful programs to pure System F, in the toplevel region our implementation supports all native effects present in the target language, like native mutable references, file IO, etc.

Our CPS translation employs standard techniques to avoid administrative β- and η-redexes [6, 34], is curried [13, 36], but not fully iterated [34, 35], which is to say that we do not abstract more than one continuation in function definitions. Rather, additional continuation parameters are added by instantiating the answer type with another layer of CPS as required.

Effekt has higher-rank polymorphism originating from function parameters as well as from polymorphic effect signatures (*e.g.*, `effect Exc { def raise[A](): A }`). Since SML is a language with a Hindley-Milner-style [16, 25] type system, it does not support higher-rank polymorphism. Due to this limitation, our SML backend currently does neither support higher-rank function types nor polymorphic effect signatures.

Another problem is that the translation from Λ_{cap} to System F presented by Schuster et al. [36] makes heavy use of higher-rank types. Region abstractions are translated to type abstractions which makes region-polymorphic function parameters have a higher-rank type. Moreover, the type of subregion evidence $\rho_1 \sqsubseteq \rho_2$ is translated to $\forall a.\ \textsc{Cps}\ \mathcal{T}[\![\ \rho_2\ ]\!]\ a \to \textsc{Cps}\ \mathcal{T}[\![\ \rho_1\ ]\!]\ a$. Functions that take evidence parameters, which are virtually all translated functions, have rank-2 type. While this might sound like a severe limitation, we observed that in many functions evidence is not actually used but only passed on and thus can often be treated parametrically. When evidence is actually used, its type argument is often inferred monomorphically. However, using the same evidence at two different types in HM will lead to a type mismatch, a problem not present in System F.

In order to admit more Effekt programs to be translated to SML, we perform evidence monomorphization, where we effectively partially evaluate programs with respect to evidence and specialize effect handler implementations to the region they are called in. We consider both, the limitation of SML not supporting higher-rank types, as well as the implemented evidence monomorphization as non-essential aspects of the present paper. We could have chosen an arbitrary different target platform that does support higher-rank types.

4.2 Benchmarks

One of our goals was to reproduce the performance results of Schuster et al. [35] in a realistic source-level language, in particular, their conjecture that specialized optimizations or special reduction theories are not needed to remove abstraction overhead; rather, existing optimizing compilers can do the job. In Table 1 we present the results of measuring the running time of programs written in Effekt and compiled to our SML backend against the running time of the same programs written in other languages with effect handlers. The benchmark programs are taken from a community benchmark suite that has been designed specifically for effect handler implementations [15]. The repository contains detailed explanations for each of the benchmark programs. Benchmarks were conducted on a 12th Gen Intel(R) Core(TM) i7-1255U running Ubuntu 22.04.

Table 1 Benchmark results comparing Eff, OCaml, Koka, our implementation of lift inference in Effekt, and a hand-optimized baseline. Fastest mean for each benchmark is highlighted in gray.

	Mean time in ms (standard deviation)				
Benchmark	**Eff**	**OCaml**	**Koka**	**Effekt**	**Baseline**
Countdown (200M)	72.0 (±13.2)	1976.1 (±26.6)	1598.0 (±24.0)	44.5 (±1.0)	**44.5** (±0.9)
Fibonacci (42)	**1093.6** (±5.5)	1161.5 (±12.0)	1222.6 (±27.0)	1335.4 (±12.0)	1335.4 (±24.5)
Product Early (100k)	535.7 (±71.9)	**113.0** (±0.4)	1506.6 (±20.0)	238.2 (±33.4)	113.0 (±0.9)
Iterator (40M)	516.3 (±17.6)	195.4 (±1.3)	1082.0 (±9.6)	92.5 (±10.7)	**13.7** (±0.5)
Queens (12)	262.2 (±6.1)	635.6 (±1.8)	2643.5 (±26.6)	117.2 (±0.3)	**96.6** (±1.0)
Tree Explore (16)	161.1 (±3.8)	**142.9** (±2.2)	278.4 (±5.1)	187.1 (±4.4)	179.1 (±2.0)
Triples (300)	125.0 (±4.4)	315.5 (±3.3)	2635.8 (±11.4)	30.0 (±0.5)	**25.1** (±0.4)
Resume Non-tail (10k)	182.4 (±15.9)	190.4 (±1.0)	1601.5 (±16.6)	85.9 (±3.5)	**62.5** (±3.0)
Parsing (20k)	2061.7 (±177.3)	1443.5 (±14.6)	3220.4 (±253.6)	88.6 (±0.8)	**88.1** (±1.0)

4.2.1 Systems

We compare Eff, Multicore OCaml, and Koka with our own implementation in Effekt. Eff [32, 18] is a language with effect handlers and specific optimizations for those. After optimization it generates OCaml code which is then compiled with ocamlopt 4.14.1. Unfortunately, investigation of the generated code reveals that in many programs the specialization of functions to handlers is not triggered. Multicore OCaml [8] 4.12.0 is a language with effect handlers and a very fast runtime. While it does not officially support multiple resumptions, which some benchmarks use, it has limited support for those which is sufficient to run these benchmarks. Koka [43] is a language with effect handlers, a fast runtime, and an optimizing compiler. The Koka compiler generates C code which then is further compiled with gcc. Our own Effekt compiler produces code in Standard ML and then uses MLton 20210117 to compile it. We instruct MLton to choose numbers to be 64bit integers to match the behavior of the other languages. Finally, as a baseline, we have taken the programs produced by our Effekt compiler and minimized and hand-optimized them using native effects where possible.

4.2.2 Results

Our findings (presented in Table 1) are generally positive. Effekt outperforms the other languages in most benchmarks, sometimes by an order of magnitude. Speedups range from around 1.6x–16.3x to the next best system for each particular benchmark. On the other side, we only observe slowdowns of 1.2x–2.1x compared to the best system for the specific benchmark. Our benchmarks are available as an artifact (see Section 7).

The Countdown benchmark uses the state effect to tail-recursively count down from a given number. Some implementations (OCaml and Koka) use references to implement the state effect, others (such as Eff and ours after evidence monomorphization) modify the answer type to be a function taking the state. In OCaml and Koka getting and setting the state goes through performing an effect operation, while Eff and Effekt are able to optimize this indirection away.

The Fibonacci benchmark does not actually use effect handlers. Eff, Koka, and Effekt generate special code for pure functions and the performance is competitive. The code generated by Eff and Effekt (after MLton optimizations) is very similar to the handwritten direct-style OCaml code and runtime differences amount to the different language runtimes used to execute the code.

The Product Early benchmark pushes 1,000 frames onto the stack and then discards all of them by throwing an exception. We can see a slowdown compared to native exceptions in OCaml and in the MLton baseline. This is due to the fact that the implementations of exceptions in both runtimes are very efficient and indeed faster than our approach of translating to CPS.

The Iterator benchmark models push streams by using an effect to emit values. The handler uses state to add all values and calls the continuation in tail position. We can see speedups compared to OCaml of around 2.1x. The optimizations performed by Eff seem to be blocked and thus the handler is not optimized away. We expect this problem to be technical and not fundamental in their approach.

The Queens benchmark searches for a solution to place n queens on a chessboard. It heavily uses continuations in a non-trivial way to perform backtracking search. We can observe a speedup of 2.2x over Eff, but note again that their optimizations seem to be blocked.

The Tree Explore benchmark constructs a tree and then traverses it to collect all leaves.

It uses a choice effect to simulate a non-deterministic traversal. We can see a slowdown of 1.3x compared to OCaml. The reason for this is again the difference between OCaml and MLton. Indeed, we have translated the code we generate from Standard ML to OCaml and observed that it runs faster than the OCaml variant using native effects.

The Triples benchmark makes heavy use of continuations to perform a backtracking search. We see speedups of around 4.2x compared to Eff, but yet again note that the rewrite rules of Karachalias et al. [18] seem not to be applied fully.

The Resume Non-tail benchmark calls an effect operation in a loop. The handler resumes in non-tail position and thus aggregates N stack frames. After the loop returns, the stack frames are popped one-after-another. Again, we can see speedups of 2.1x over Eff, where the rewrites are not fully applied.

Finally, the Parsing benchmark defines a streaming parser which uses three effects:

```
def parse(a: Int): Unit / {Read, Emit, Stop} = ...
```

The `Read` effect reads a character, the `Emit` effect emits the result of parsing a line, and the `Stop` effect stops when an unrecognized character is found. The function `parse` is used under three handlers, one for each of the effects:

```
sum { catch { feed(n) { parse(0) } } }
```

The program has non-trivial control flow, which is abstracted away by the use of effect handlers. Moreover we could use the same function `parse` with a different source of characters and a different target of emitted values. Our Effekt implementation significantly outperforms the other languages by a factor of 16.3x–36.3x. It is the only benchmark that relies on evidence monomorphization in order to compile. Our implementation specializes this function to the handlers surrounding it, which after optimization results in a single tight loop, which is exactly our original goal: to remove all abstraction overhead introduced by using effect handlers.

In general, our approach works better in the cases where effects and resumptions are used extensively. In these cases we observe large speedups over the other implementations. That said, the optimizations for Eff were often blocked in these benchmarks. We would expect the results for Eff to be much closer to ours, if the optimizations kick in. Our results are often quite close to the hand-optimized baseline. In these cases our implementation, of which lift-inference is an integral part, is able to remove all abstraction introduced by the use of effect handlers to structure the program. In the other cases, more investigation is needed in order to remove the gap between compiled code using effect handlers and hand-optimized code using native effects.

5 Related Work

We have presented a translation from System Ξ, a calculus with second-class capabilities to Λ_{cap}, a calculus with region-based effects. In combination with a translation to iterated CPS this enables efficient compilation of effect handlers. In this section, we compare our approach to existing work.

5.1 Efficient Compilation of Effect Handlers

Closely related is the work on explicit effect subtyping for algebraic effect handlers in Eff [32]. Their main motivation is to use this explicit information in the optimization of programs using effect handlers [18]. In particular, they define source-to-source rewrite rules on an intermediate representation called ExEff. The rewrite rules are designed to propagate handlers down until they reach an effect operation in which case the effect operation can be statically reduced. Intermediate frames are aggregated in the return clause of the handler. While our motivation is ultimately the same, our work is in the context of *lexical* effect handlers as they appear in Effekt. Also, we do not define a reduction theory on a language with effects and handlers, but instead (via Λ_{cap}) define a translation to System F in CPS. This way, existing optimizing compilers for functional languages (such as SML) can readily be used. Karachalias et al. [18] support first-class functions which makes their language more general than ours. While their explicitly typed language applies subtyping coercions to arbitrary computations, we pass evidence values along and only use them at effect operations where they are needed.

Also highly related in this regard is the work on evidence passing [44, 43], which provides the basis for the implementation of effect handlers in the Koka language. The idea is to pass evidence vectors down to effect operations. These evidence vectors consist of pairs of labels and handler implementations so that handlers can be looked up in place. In contrast, evidence in Λ_{cap} is just a list of labels and the handler implementations are passed as capabilities. Evidence passing is defined in the context of *dynamically* scoped handlers and hence does not reflect the *lexical* nesting of handlers as regions and subregion evidence in Λ_{cap} do. Xie et al. [44] define an evidence passing translation from an algebraic effect calculus to an evidence calculus, thus determining the evidence vectors statically. This translation is facilitated by the effect system of the algebraic effect calculus based on rows of effect labels. In contrast, our source calculus System Ξ does not feature a visible effect system and instead relies on second-class capabilities. Xie and Leijen [43] do not define a translation but achieve evidence passing by defining appropriate evaluation rules for the algebraic effect calculus, hence their evidence vectors are created dynamically. This allows them to lift the restriction of scoped resumptions, imposed by Xie et al. [44]. Both approaches support first-class functions. Moreover, both papers define a translation to System F^{v}, a polymorphic lambda calculus with support for multi-prompt monads. Instead, by building on Schuster et al. [36], we directly compile to System F in CPS.

5.2 Languages with Second-Class Values

Our lift inference consumes programs in System Ξ [4], a language with second-class functions and capabilities. It is inspired by the work of Osvald et al. [28] who present $\lambda^{1/2}$ a lambda calculus that features both first-class and second-class functions, but no control effects. Their work in turn builds on type-based escape analysis [12]. In $\lambda^{1/2}$, second-class functions cannot be returned, nor closed over by first-class functions. In contrast, System Ξ does not support first-class functions and in consequence our translation does not have to handle them. We do not expect any complications in extending System Ξ and our translation to first-class functions in the style of $\lambda^{1/2}$—that is, to first-class functions that cannot close over second-class functions and capabilities. As they do not contain capabilities that need to satisfy some subregion constraint, they can always run in the toplevel region, so we could just pretend that one to be their definition-site region.

Brachthäuser et al. [5] present System $\mathcal{C}$ as an extension of System Ξ to support a fine-grained notion of second-class values. Their calculus introduces explicit `box` and `unbox` constructs, inspired by modal logics. They also extend the type system to track which capabilities are used by a statement or block. Boxing takes a second-class block and turns it into a first-class value, where the type of the boxed block specifies the necessary capabilities (*e.g.*, `Int ⇒ Int at {yield}`). To call a boxed function it needs to be unboxed first. When unboxing, the type system ensures that the necessary capabilities are still available, preventing functions from closing over capabilities and then leaving the scope of a handler. This system is more expressive than $\lambda^{1/2}$ and it is less clear to us how to translate the sets of capabilities (*e.g.*, `{yield}`) to the corresponding region. We leave studying the translation of System $\mathcal{C}$ to Λ_{cap} to future work.

Xhebraj et al. [42] present another variant of $\lambda^{1/2}$, called $\lambda^{1/2}_{\hookleftarrow}$, which supports returning second-class functions and is designed to be used for stack allocating memory. Safety is achieved by modifying the runtime semantics: When a second-class value is returned, the returning frame is simply not removed. While this is an elegant solution, our goal is to target standard runtime systems like System F.

5.3 Languages with Regions and Subregioning

Our lift inference produces programs in Λ_{cap} [36] featuring explicit regions and subregion evidence. We use a generalization which allows for non-scoped continuations. While we follow Schuster et al. [36] and use regions and evidence to track the lexical nesting of handlers on the stack, the original usage of regions is in memory management [40] and more generally resource management. Our notion of regions could in principle also be used for resource management. A recent calculus in this regard which is close to Λ_{cap} is presented by Schuster et al. [37]. They use regions and subregion evidence to deal with the management of resources in the presence of exception handlers. In contrast to this work, they do not deal with general effect handlers and do not consider inference of regions and evidence. The work of Schuster et al. [37] is based on Kiselyov and Shan [20], who also perform region inference. Their approach, however, is very different from ours, as they encode regions using monad transformers and hence rely on type inference to infer regions. Likewise, other algorithms [40, 41] for region inference in the context of memory management are different from ours, often creating fresh regions for each variable and subsequently analysing which of them can be unified. In contrast to this prior work, we infer regions by establishing a connection between the lexical scoping of second-class values and regions.

It might be especially interesting to consider our CPS translation with a target language which already has a notion of regions, like the ML Kit [41], and potentially try to preserve region annotations. We leave exploration of this to future work.

5.4 Dictionary Passing and Monad Polymorphism in Haskell

In Haskell it is typical to use stacks of monad transformers [23] and type classes to compose different programs using different effects into one [17]. When type classes are implemented by dictionary passing, this is not unlike our passing of capabilities, but implicit. When effect operations corresponding to a lower layer in the monad-transformer stack are used, they have to be lifted through all layers above, just like in our work. Finding the correct composition of lifts is automatic and works by type class resolution guided by the type of computations. In contrast to this, we assume that capabilities are passed explicitly and find the correct composition of evidence by a transformation guided by program terms. This has the advantage that different instances of the same effect are easily disambiguated by passing different capabilities.

This problem was observed by Figueroa et al. [11] and named effect interference. For them, the interference between different instances of the same effect is also a security concern. As a solution they propose the explicit passing of capabilities. However, their notion of capability is different from ours in that they use them to ensure certain security guarantees on top of monad transformers while we use them as an implementation technique for effect handlers. Consequently, our capabilities contain the operation to execute when they are used while theirs do not.

Schrijvers et al. [33] compare and contrast monad transformers and the traditional implementation of dynamic effect handlers in terms of a free monad in Haskell. There, the problem of lift inference manifests in a different way. Effectful programs are written against an open union of signatures [19]. The challenge is for a given effect operation to find the correct injection into this open union. Again, Haskell type classes can be used for this to some extent.

However, using type class resolution to find the correct handler is problematic when there are multiple instances of the same effect in the same program. As a solution, Devriese [7] propose explicit passing of type class dictionaries, which essentially are what we call capabilities. Moreover, in order to nest handlers, they propose the explicit use of liftings, essentially what we call evidence. They also present a case study that speaks for the feasibility of their explicit approach. We, however, infer the correct use of evidence, so programmers do not have to do so explicitly. In their setting liftings are general monad morphisms, while in our setting we specialize them to the continuation- and state monads. Another difference is that they apply these liftings to capabilities, while we pass evidence to the places where capabilities are used.

6 Conclusion

In this paper, we have presented a way to infer lifting information for lexical effects and handlers, by giving a typed translation from a calculus System Ξ with second-class capabilities to a calculus Λ_{cap} with explicit regions and subregion evidence. Our translation preserves typability and semantics. It makes use of the second-class property to provide a clear connection for the definition-site and each call-site of a function. This establishes a precise relation between reasoning based on the second-class property and region-based regioning.

Moreover, we have evaluated the implications of lift inference practically, by implementing it as a compiler phase for a source-level language. To this end, we have further implemented the CPS translation for Λ_{cap} described by Schuster et al. [36], which makes heavy use of the information provided by lift inference to enable efficient compilation of effect handlers. Our benchmarks indicate that our approach is competitive with other state-of-the-art implementations of effect handlers and often outperforms them.

While the second-class property of our source calculus is particularly helpful for lift inference, it can sometimes be a restriction in programming. In the future, it would be interesting to investigate whether it is possible to extend our approach to a language which has a controlled way of using effectful first-class functions, as described, *e.g.*, by Brachthäuser et al. [5]. Furthermore, it would be interesting to see how lift inference can be performed for languages with traditional effect handlers. Also, while System Ξ uses types to enforce the second-class property for capabilities, we believe that any mechanism to enforce this would do. But we leave a fully precise exploration of this for future work.

7 Data-Availability Statement

The benchmarks from Section 4 are available to be run in a Docker container in the accompanying artifact [27].

Acknowledgments

We thank Matthew Fluet for his helpful assistance with MLton. The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

References

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

2 D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL): 8:1–8:30, Dec. 2017. ISSN 2475-1421. doi: 10.1145/3158096.

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

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

5 J. I. Brachthäuser, P. Schuster, E. Lee, and A. Boruch-Gruszecki. Effects, capabilities, and boxes: From scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.*, 6(OOPSLA), apr 2022. doi: 10.1145/3527320.

6 O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. doi: 10.1017/S0960129500001535.

7 D. Devriese. Modular effects in haskell through effect polymorphism and explicit dictionary applications: A new approach and the μverifast verifier as a case study. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 1–14, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342589. URL `https://doi.org/10.1145/3331545.3342589`.

8 S. Dolan, L. White, and A. Madhavapeddy. Multicore OCaml. In *OCaml Workshop*, 2014.

9 R. K. Dybvig, S. Peyton Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, Nov. 2007. ISSN 0956-7968. doi: 10.1017/S0956796807006259.

10 K. Farvardin and J. Reppy. From folklore to fact: Comparing implementations of stacks and continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 75–90, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385994. URL `https://doi.org/10.1145/3385412.3385994`.

11 I. Figueroa, N. Tabareau, and E. Tanter. Effect capabilities for haskell: Taming effect interference in monadic programming. *Science of Computer Programming*, 119, Nov 2015. doi: 10.1016/j.scico.2015.11.010.

12 J. Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, May 1998. doi: 10.1017/S0956796898003025.

13 D. Hillerström, S. Lindley, B. Atkey, and K. Sivaramakrishnan. Continuation passing style for effect handlers. In *Formal Structures for Computation and Deduction*, volume 84 of *LIPIcs*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

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

15 D. Hillerström, F. Koprivec, and P. Schuster (benchmarking chairs). Effect handlers benchmarks suite. 2023. URL `https://github.com/effect-handlers/effect-handlers-bench`.

16 J. Hindley. The principal type scheme of an object in combinatory logic. *Trans. of the American Mathematical Society*, 146:29–60, Dec. 1969. doi: 10.2307/1995158.

17 M. P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 97–136, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. doi: 10.1007/3-540-59451-5_4.

18 G. Karachalias, F. Koprivec, M. Pretnar, and T. Schrijvers. Efficient compilation of algebraic effect handlers. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi: 10.1145/3485479. URL `https://doi.org/10.1145/3485479`.

19 O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Proceedings of the Haskell Symposium*, pages 94–105, New York, NY, USA, 2015. ACM. doi: 10.1145/2887747.2804319.

20 O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *Proceedings of the Haskell Symposium*, Haskell '08, New York, NY, USA, 2008. ACM. doi: 10.1145/1411286.1411288.

21 D. Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 486–499, New York, NY, USA, 2017. ACM. doi: 10.1145/3093333.3009872.

22 P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003. ISSN 0890-5401. doi: 10.1016/S0890-5401(03)00088-9. URL `https://doi.org/10.1016/S0890-5401(03)00088-9`.

23 S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 333–343, New York, NY, USA, 1995. ACM. doi: 10.1145/199448.199528.

24 S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 500–514, New York, NY, USA, 2017. ACM. doi: 10.1145/3009837.3009897.

25 R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:248–375, 1978. doi: 10.1016/0022-0000(78)90014-4.

26 R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of standard ML: revised*. MIT press, 1997. doi: 10.7551/mitpress/2319.001.0001.

27 M. Müller, P. Schuster, J. L. Starup, K. Ostermann, and J. I. Brachthäuser. Artifact of the paper 'From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers', Sept. 2023.

28 L. Osvald, G. Essertel, X. Wu, L. I. G. Alayón, and T. Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 234–251, New York, NY, USA, 2016. ACM. doi: 10.1145/3022671.2984009.

29 G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer-Verlag, 2009. doi: 10.1007/978-3-642-00590-9_7.

30 G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. doi: 10.2168/LMCS-9(4:23)2013.

31 M. Pretnar, A. H. S. Saleh, A. Faes, and T. Schrijvers. Efficient compilation of algebraic effects and handlers. Technical report, Department of Computer Science, KU Leuven; Leuven, Belgium, 2017.

32 A. H. Saleh, G. Karachalias, M. Pretnar, and T. Schrijvers. Explicit effect subtyping. In A. Ahmed, editor, *Programming Languages and Systems*, pages 327–354, Cham, Switzerland, 2018. Springer International Publishing. ISBN 978-3-319-89884-1. doi: 10.1007/978-3-319-89884-1_12.

33 T. Schrijvers, M. Piróg, N. Wu, and M. Jaskelioff. Monad transformers and modular algebraic effects: What binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 98–113, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342595. URL `https://doi.org/10.1145/3331545.3342595`.

34 P. Schuster and J. I. Brachthäuser. Typing, representing, and abstracting control. In *Proceedings of the Workshop on Type-Driven Development*, pages 14–24, New York, NY, USA, 2018. ACM. doi: 10.1145/3240719.3241788.

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

36 P. Schuster, J. I. Brachthäuser, M. Müller, and K. Ostermann. A typed continuation-passing translation for lexical effect handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 566–579, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523710.

37 P. Schuster, J. I. Brachthäuser, and K. Ostermann. Region-based resource management and lexical exception handlers in continuation-passing style. In I. Sergey, editor, *Programming Languages and Systems*, pages 492–519, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99336-8. doi: 10.1007/978-3-030-99336-8_18.

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

39 H. Thielecke. From control effects to typed continuation passing. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 139–149, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136285. doi: 10.1145/604131.604144.

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

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

42 A. Xhebraj, O. Bračevac, G. Wei, and T. Rompf. What If We Don't Pop the Stack? The Return of 2nd-Class Values. In K. Ali and J. Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-225-9. doi: 10.4230/LIPIcs.ECOOP.2022.15.

43 N. Xie and D. Leijen. Generalized evidence passing for effect handlers: Efficient compilation of effect handlers to c. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi: 10.1145/3473576. URL `https://doi.org/10.1145/3473576`.

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

45 Y. Zhang and A. C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL):5:1–5:29, Jan. 2019. ISSN 2475-1421. doi: 10.1145/3290318.

A Proofs for Lift-Inference Translation from System Ξ to Λ_{cap}

In this appendix, we give the proofs of typability and semantics preservation for the lift-inference translation. We use pr_i for the i-th projection on a pair.

A.1 Typability Preservation

We start with typability preservation. First, we prove preservation of the soundness of lifting environments (Lemma 5).

▶ **Lemma 8** (Soundness of Environments for the Lift-Inference Translation). *All adaptions of lifting environments made by the lift-inference translation take sound environments to sound environments.*

Proof. By case distinction. There are three cases where the environment is changed in the translation.

case Def

Given a sound environment $\mathsf{E} = \{ \rho, \Delta, m, \Gamma_E \}$, we have to show that

$$\begin{aligned} \mathsf{E}' &:= \mathsf{E}, f \mapsto (\mathcal{R}[\![b]\!]^{\mathsf{E}} \mid \mathcal{C}[\![b]\!]^{\mathsf{E}}) \\ &= \{ \rho, (\Delta, f), (m, f \mapsto (\mathcal{R}[\![b]\!]^{\mathsf{E}} \mid \mathcal{C}[\![b]\!]^{\mathsf{E}})), \Gamma_E \} \end{aligned}$$

is sound, too.
For $g \in \mathsf{dom}(\Delta)$ we have $\mathcal{R}[\![g]\!]^{\mathsf{E}'} = \mathsf{pr}_1(m(g)) = \mathcal{R}[\![g]\!]^{\mathsf{E}}$ (we assume all bindings to be distinct) and similarly $\mathcal{C}[\![g]\!]^{\mathsf{E}'} = \mathsf{pr}_2(m(g)) = \mathcal{C}[\![g]\!]^{\mathsf{E}}$ and hence

$$\Gamma_E \vdash \mathcal{C}[\![g]\!]^{\mathsf{E}'} : \rho \sqsubseteq \mathcal{R}[\![g]\!]^{\mathsf{E}'}$$

by soundness of E.
For f note that by typing we either have $b = g$ for some $g \in \mathsf{dom}(\Delta)$ or $b = w$. In the first case the same reasoning as above applies. In the second case we have $\mathcal{R}[\![f]\!]^{\mathsf{E}'} = \mathcal{R}[\![w]\!]^{\mathsf{E}} = \rho$ and similarly $\mathcal{C}[\![f]\!]^{\mathsf{E}'} = \mathbb{0}$ and thus

$$\Gamma_E \vdash \mathcal{C}[\![f]\!]^{\mathsf{E}'} : \rho \sqsubseteq \mathcal{R}[\![f]\!]^{\mathsf{E}'}$$

by rule Reflexive.

case Try

Given a sound environment $\mathsf{E} = \{ \rho, \Delta, m, \Gamma_E \}$, we have to show that

$$\begin{aligned} \mathsf{E}' &:= \mathsf{E} \oplus n, c \mapsto (r \mid \mathbb{0}) \\ &= \{ r, (\Delta, c), (m \oplus n, c \mapsto (r \mid \mathbb{0})), (\Gamma_E, r, n : r \sqsubseteq \rho) \} \\ \mathsf{E}'' &:= \mathsf{E}, k \mapsto (\rho \mid \mathbb{0}) \\ &= \{ \rho, (\Delta, k), (m, k \mapsto (\rho \mid \mathbb{0})), \Gamma_E \} \end{aligned}$$

are sound.
First consider E'. Note that $\Gamma_E, r, n : r \sqsubseteq \rho \vdash \mathbb{0} : r \sqsubseteq r$ by rule Reflexive which is what we need for c.
For $f \in \mathsf{dom}(\Delta)$ we have $\mathcal{R}[\![f]\!]^{\mathsf{E}'} = \mathsf{pr}_1((m \oplus n)(f)) = \mathsf{pr}_1(m(f)) = \mathcal{R}[\![f]\!]^{\mathsf{E}}$ and $\mathcal{C}[\![f]\!]^{\mathsf{E}'} = \mathsf{pr}_2((m \oplus n)(f)) = n \oplus \mathcal{C}[\![f]\!]^{\mathsf{E}}$. By soundness of E (and weakening) we have

$$\Gamma_E, r, n : r \sqsubseteq \rho \vdash \mathcal{C}[\![f]\!]^{\mathsf{E}} : \rho \sqsubseteq \mathcal{R}[\![f]\!]^{\mathsf{E}'}$$

and rule EviVar yields

$$\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho \vdash\ n\ :\ r\ \sqsubseteq\ \rho$$

Hence, we can apply rule Transitive to obtain

$$\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho \vdash\ n\ \oplus \mathcal{C}[\![\ f\ ]\!]^{\mathsf{E}}\ :\ r\ \sqsubseteq\ \mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}'}$$

Now consider E''. For k, rule Reflexive yields $\Gamma_E \vdash\ \mathbb{0}\ :\ \rho\ \sqsubseteq\ \rho$.
For $g\ \in \mathsf{dom}(\Delta)$ we have $\mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}''}\ =\ \mathsf{pr}_1(m(g))\ =\ \mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}}$ and similarly $\mathcal{C}[\![\ g\ ]\!]^{\mathsf{E}''}\ =\ \mathcal{C}[\![\ g\ ]\!]^{\mathsf{E}}$ and hence

$$\Gamma_E \vdash\ \mathcal{C}[\![\ g\ ]\!]^{\mathsf{E}''}\ :\ \rho\ \sqsubseteq\ \mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}''}$$

by soundness of E.

case Block

Given a sound environment $\mathsf{E}\ =\ \{\ \rho,\ \Delta,\ m,\ \Gamma_E\ \}$, we have to show that

$$\begin{aligned}\mathsf{E}'\ &:=\ \mathsf{E}\ \oplus n,\ \overline{f \mapsto (r_f\ \mid n_f)},\ \overline{r_f,\ n_f:\ r\ \sqsubseteq\ r_f}\\ &=\ \{\ r,\ (\Delta,\ \overline{f}),\ (m\ \oplus n,\ \overline{f \mapsto (r_f\ \mid n_f)}),\ (\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho,\ \overline{r_f,\ n_f:\ r\ \sqsubseteq\ r_f})\ \}\end{aligned}$$

is sound. For the $\overline{f}$ we have $\overline{\mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}'}\ =\ r_f}$ and $\overline{\mathcal{C}[\![\ f\ ]\!]^{\mathsf{E}'}\ =\ n_f}$, hence rule EviVar gives us

$$\overline{\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho,\ \overline{r_f,\ n_f:\ r\ \sqsubseteq\ r_f} \vdash\ \mathcal{C}[\![\ f\ ]\!]^{\mathsf{E}'}\ :\ r\ \sqsubseteq\ \mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}'}}$$

For $g\ \in \mathsf{dom}(\Delta)$ we have $\mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}'}\ =\ \mathsf{pr}_1((m\ \oplus n)(g))\ =\ \mathsf{pr}_1(m(g))\ =\ \mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}}$ and $\mathcal{C}[\![\ g\ ]\!]^{\mathsf{E}'}\ =\ \mathsf{pr}_2((m\ \oplus n)(g))\ =\ n\ \oplus \mathcal{C}[\![\ g\ ]\!]^{\mathsf{E}}$. By soundness of E (and weakening) we have

$$\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho,\ \overline{r_f,\ n_f:\ r\ \sqsubseteq\ r_f} \vdash\ \mathcal{C}[\![\ g\ ]\!]^{\mathsf{E}}\ :\ \rho\ \sqsubseteq\ \mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}'}$$

and rule EviVar yields

$$\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho,\ \overline{r_f,\ n_f:\ r\ \sqsubseteq\ r_f} \vdash\ n\ :\ r\ \sqsubseteq\ \rho$$

Hence, we can apply rule Transitive to obtain

$$\Gamma_E,\ r,\ n:\ r\ \sqsubseteq\ \rho,\ \overline{r_f,\ n_f:\ r\ \sqsubseteq\ r_f} \vdash\ n\ \oplus \mathcal{C}[\![\ g\ ]\!]^{\mathsf{E}}\ :\ r\ \sqsubseteq\ \mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}'}$$

◀

Now we prove typability preservation for the lift-inference translation (Theorem 6).

▶ **Theorem 9** (Typability Preservation for the Lift-Inference Translation)**.**
For $\mathsf{E}\ =\ \{\ \rho,\ \Delta,\ m,\ \Gamma_E\ \}$ sound,
if $\Gamma\ \mid\ \Delta \vdash\ s\ :\ \tau$, then $\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\ \Delta\ ]\!]^{E}\ \mid\ \rho \vdash\ \mathcal{S}[\![\ s\ ]\!]^{E}\ :\ \tau$;
if $\Gamma\ \mid\ \Delta \vdash\ b\ :\ \sigma$, then $\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\ \Delta\ ]\!]^{E} \vdash\ B[\![\ b\ ]\!]^{E}\ :\ \mathcal{T}[\![\ \sigma\ ]\!]_{\mathcal{R}[\![\ b\ ]\!]^{E}}$;
if $\Gamma \vdash\ v\ :\ \tau$, then $\Gamma_E,\ \Gamma \vdash\ v\ :\ \tau$.

Proof. Induction over the typing derivation of the term in System Ξ. We make implicit use of the structural rules.

case Lit

Immediate.

case Var

Given $\Gamma \vdash x : \tau$, we have $\Gamma(x) = \tau$ and hence $\Gamma_E,\ \Gamma \vdash x : \tau$ by rule Var.

case BlockVar
Given $\Gamma \mid \Delta \vdash f : \sigma$, we have $\Delta(f) = \sigma$. Thus, $\mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}}(f) = \mathcal{T}[\![\ \sigma\ ]\!]_{\mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}}}$ and hence
$\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}} \vdash f : \mathcal{T}[\![\ \sigma\ ]\!]_{\mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}}}$ by rule Var.

case Block
Given $\Gamma \mid \Delta \vdash \{\ (\overline{x : \tau},\ \overline{f : \sigma}) \Rightarrow s_0\ \} : (\overline{\tau},\ \overline{\sigma}) \to \tau_0$, we have $\Gamma,\ \overline{x : \tau} \mid \Delta,\ \overline{f : \sigma} \vdash s_0 : \tau_0$. We need to show that

$$\begin{aligned}&\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}} \vdash \\ &\quad \{\ [r,\ \overline{r_f};\ n : r \sqsubseteq \rho,\ \overline{n_f : r \sqsubseteq r_f}](\overline{x : \tau},\ \overline{f : \mathcal{T}[\![\ \sigma\ ]\!]_{r_f}})\ \textbf{at}\ r \Rightarrow \mathcal{S}[\![\ s_0\ ]\!]^{\mathsf{E}'}\ \} : \\ &\quad\ \forall[r,\ \overline{r_f};\ r \sqsubseteq \rho,\ \overline{r \sqsubseteq r_f}]\ (\overline{\tau},\ \overline{\mathcal{T}[\![\ \sigma\ ]\!]_{r_f}}) \to r\ \tau_0\end{aligned}$$

where $r,\ n,\ \overline{r_f,\ n_f}$ are fresh and

$$\begin{aligned}\mathsf{E}' &:= \mathsf{E} \oplus n,\ \overline{f \mapsto (r_f \mid n_f)},\ \overline{r_f,\ n_f : r \sqsubseteq r_f} \\ &= \{\ r,\ (\Delta,\ \overline{f}),\ (m \oplus n,\ \overline{f \mapsto (r_f \mid n_f)}),\ (\Gamma_E,\ r,\ n : r \sqsubseteq \rho,\ \overline{r_f,\ n_f : r \sqsubseteq r_f})\ \}\end{aligned}$$

To be able to invoke rule Fun we hence need

$$\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}},\ r,\ \overline{r_f},\ n : r \sqsubseteq \rho,\ \overline{n_f : r \sqsubseteq r_f},\ \overline{x : \tau},\ \overline{f : \mathcal{T}[\![\ \sigma\ ]\!]_{r_f}} \mid r \vdash \mathcal{S}[\![\ s_0\ ]\!]^{\mathsf{E}'} : \tau_0$$

The assertion for statements instantiated with E' (which is applicable since E' is sound by Lemma 5 and $\mathsf{E}'.\Delta = \Delta,\ \overline{f}$) gives us

$$\Gamma_E,\ r,\ n : r \sqsubseteq \rho,\ \overline{r_f,\ n_f : r \sqsubseteq r_f},\ \Gamma,\ \overline{x : \tau},\ \mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}'},\ \overline{f : \mathcal{T}[\![\ \sigma\ ]\!]_{\mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}}}} \mid r \vdash \mathcal{S}[\![\ s_0\ ]\!]^{\mathsf{E}'} : \tau_0$$

But for $g \in \Delta$ we have $\mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}'} = \mathsf{pr}_1((m \oplus n)(g)) = \mathsf{pr}_1(m(g)) = \mathcal{R}[\![\ g\ ]\!]^{\mathsf{E}}$, hence, we have
$\mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}} = \mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}'}$. And since we also know that $\overline{\mathcal{R}[\![\ f\ ]\!]^{\mathsf{E}} = r_f}$, we have what we need.

case Val
Given $\Gamma \mid \Delta \vdash \textbf{val}\ x = s_0;\ s : \tau$, we have $\Gamma \mid \Delta \vdash s_0 : \tau_0$ and $\Gamma,\ x : \tau_0 \mid \Delta \vdash s : \tau$ for some τ_0. By the induction hypothesis we have $\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}} \mid \rho \vdash \mathcal{S}[\![\ s_0\ ]\!]^{\mathsf{E}} : \tau_0$ and
$\Gamma_E,\ \Gamma,\ x : \tau_0,\ \mathcal{T}[\![\ \Delta\ ]\!]^{\mathsf{E}} \mid \rho \vdash \mathcal{S}[\![\ s\ ]\!]^{\mathsf{E}} : \tau$. Hence we obtain

$$\dfrac{\Gamma_E, \Gamma, \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \mid \rho \vdash \mathcal{S}[\![s_0]\!]^{\mathsf{E}} : \tau_0 \qquad \Gamma_E, \Gamma, \mathcal{T}[\![\Delta]\!]^{\mathsf{E}}, x : \tau_0 \mid \rho \vdash \mathcal{S}[\![s]\!]^{\mathsf{E}} : \tau}{\Gamma_E, \Gamma, \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \mid \rho \vdash \textbf{val}\ x = \mathcal{S}[\![s_0]\!]^{\mathsf{E}};\ \mathcal{S}[\![s]\!]^{\mathsf{E}} : \tau}\ \textsc{Val}$$

case Ret
Given $\Gamma \mid \Delta \vdash \textbf{return}\ v : \tau$, we have $\Gamma \vdash v : \tau$. By the assertion for values we have $\Gamma_E,\ \Gamma \vdash v : \tau$. Hence we obtain

$$\dfrac{\Gamma_E, \Gamma \vdash v : \tau}{\Gamma_E, \Gamma, \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \mid \rho \vdash \textbf{return}\ v : \tau}\ \textsc{Ret}$$

case Def

Given $\Gamma \mid \Delta \vdash \textbf{def}\ f = b;\ s\ : \tau$, we have $\Gamma \mid \Delta \vdash b\ : \sigma$ and $\Gamma \mid \Delta,\ f : \sigma \vdash s\ : \tau$ for some σ. By Lemma 5 we know that

$$\begin{aligned} \mathsf{E}' &:= \mathsf{E},\ f \mapsto (\mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}} \mid \mathcal{C}\llbracket\ b\ \rrbracket^{\mathsf{E}}) \\ &= \{\ \rho,\ (\Delta,\ f),\ (m,\ f \mapsto (\mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}} \mid \mathcal{C}\llbracket\ b\ \rrbracket^{\mathsf{E}})),\ \Gamma_E\ \} \end{aligned}$$

is sound. Using $\mathsf{E}'.\Delta = \Delta,\ f$, we can use the assertion for blocks and instantiate the induction hypothesis with E' to obtain

$$\begin{aligned} &\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} \vdash B\llbracket\ b\ \rrbracket^{\mathsf{E}}\ : \mathcal{T}\llbracket\ \sigma\ \rrbracket_{\mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}} \quad \text{and} \\ &\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}'},\ f : \mathcal{T}\llbracket\ \sigma\ \rrbracket_{\mathcal{R}\llbracket\ f\ \rrbracket^{\mathsf{E}'}} \mid \rho \vdash \mathcal{S}\llbracket\ s\ \rrbracket^{\mathsf{E}'}\ : \tau \end{aligned}$$

respectively. For $g \in \Delta$ we have $\mathcal{R}\llbracket\ g\ \rrbracket^{\mathsf{E}'} = \mathsf{pr}_1(m(g)) = \mathcal{R}\llbracket\ g\ \rrbracket^{\mathsf{E}}$ and thus $\mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} = \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}'}$. Using $\mathcal{R}\llbracket\ f\ \rrbracket^{\mathsf{E}'} = \mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}$, we hence obtain

$$\dfrac{\dfrac{\Gamma_E, \Gamma, \mathcal{T}\llbracket\Delta\rrbracket^{\mathsf{E}} \vdash B\llbracket b\rrbracket^{\mathsf{E}}\ : \mathcal{T}\llbracket\sigma\rrbracket_{\mathcal{R}\llbracket b\rrbracket^{\mathsf{E}}}}{\Gamma_E, \Gamma, \mathcal{T}\llbracket\Delta\rrbracket^{\mathsf{E}} \mid \rho \vdash \textbf{return}\ B\llbracket b\rrbracket^{\mathsf{E}}\ : \mathcal{T}\llbracket\sigma\rrbracket_{\mathcal{R}\llbracket b\rrbracket^{\mathsf{E}}}}\,\textsc{Ret} \qquad \Gamma_E, \Gamma, \mathcal{T}\llbracket\Delta\rrbracket^{\mathsf{E}}, f : \mathcal{T}\llbracket\sigma\rrbracket_{\mathcal{R}\llbracket b\rrbracket^{\mathsf{E}}} \mid \rho \vdash \mathcal{S}\llbracket s\rrbracket^{\mathsf{E}'}\ : \tau}{\Gamma_E, \Gamma, \mathcal{T}\llbracket\Delta\rrbracket^{\mathsf{E}} \mid \rho \vdash \textbf{val}\ f = \textbf{return}\ B\llbracket b\rrbracket^{\mathsf{E}};\ \mathcal{S}\llbracket s\rrbracket^{\mathsf{E}'}\ : \tau}\,\textsc{Val}$$

case App

Given $\Gamma \mid \Delta \vdash b_0(\overline{v},\ \overline{b})\ : \tau_0$, we have $\Gamma \mid \Delta \vdash b_0 : (\overline{\tau},\ \overline{\sigma}) \to \tau_0$, $\overline{\Gamma \mid \Delta \vdash b\ : \sigma}$ and
$\overline{\Gamma \vdash v\ : \tau}$ for some $\overline{\tau},\ \overline{\sigma}$. The assertion for blocks gives us

$$\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} \vdash B\llbracket\ b_0\ \rrbracket^{\mathsf{E}}\ : \forall[r,\ \overline{r_f};\ r \sqsubseteq \mathcal{R}\llbracket\ b_0\ \rrbracket^{\mathsf{E}},\ \overline{r \sqsubseteq r_f}]\ (\overline{\tau},\ \overline{\mathcal{T}\llbracket\ \sigma\ \rrbracket_{r_f}}) \to r\ \tau_0$$

for $r,\ \overline{r_f}$ fresh. We have to show that

$$\begin{aligned} &\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} \mid \rho \vdash \\ &B\llbracket\ b_0\ \rrbracket^{\mathsf{E}}[\rho,\ \overline{\mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}};\ \mathcal{C}\llbracket\ b_0\ \rrbracket^{\mathsf{E}},\ \overline{\mathcal{C}\llbracket\ b\ \rrbracket^{\mathsf{E}}}\ ](\overline{v},\ \overline{B\llbracket\ b\ \rrbracket^{\mathsf{E}}})\ : \tau_0 \end{aligned}$$

Note that there are no free region variables in τ_0, so $\tau_0 = \tau_0[\mathfrak{sb}]$, where $\mathfrak{sb} \doteq r \mapsto \rho,\ \overline{r_f \mapsto \mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}}$. Similarly, we know that $\overline{\tau = \tau[\mathfrak{sb}]}$. Moreover, we have $\rho = r[r \mapsto \rho] = r[\mathfrak{sb}]$. And since each r_f is the only free region variable in each $\mathcal{T}\llbracket\ \sigma\ \rrbracket_{r_f}$, respectively, we further find

$$\overline{\mathcal{T}\llbracket\ \sigma\ \rrbracket_{\mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}} = \mathcal{T}\llbracket\ \sigma\ \rrbracket_{r_f}[r_f \mapsto \mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}] = \mathcal{T}\llbracket\ \sigma\ \rrbracket_{r_f}[\mathfrak{sb}]}$$

The assertion for blocks and values thus gives us

$$\overline{\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} \vdash B\llbracket\ b\ \rrbracket^{\mathsf{E}}\ : \mathcal{T}\llbracket\ \sigma\ \rrbracket_{r_f}[\mathfrak{sb}]} \quad \text{and} \quad \overline{\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} \vdash v\ : \tau[\mathfrak{sb}]}$$

To satisfy the premises for rule App the only thing left to show is

$$\begin{aligned} &\overline{\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} \vdash \mathcal{C}\llbracket\ b\ \rrbracket^{\mathsf{E}}\ : r \sqsubseteq r_f[\mathfrak{sb}]} \quad \text{and} \\ &\Gamma_E,\ \Gamma,\ \mathcal{T}\llbracket\ \Delta\ \rrbracket^{\mathsf{E}} \vdash \mathcal{C}\llbracket\ b_0\ \rrbracket^{\mathsf{E}}\ : r \sqsubseteq \mathcal{R}\llbracket\ b_0\ \rrbracket^{\mathsf{E}}[\mathfrak{sb}] \end{aligned}$$

Since none of the $r,\ r_f$ can appear in $\mathcal{R}\llbracket\ b_0\ \rrbracket^{\mathsf{E}}$ we know that $r \sqsubseteq \mathcal{R}\llbracket\ b_0\ \rrbracket^{\mathsf{E}}[\mathfrak{sb}] = \rho \sqsubseteq \mathcal{R}\llbracket\ b_0\ \rrbracket^{\mathsf{E}}$ and we further have that $\overline{r \sqsubseteq r_f[\mathfrak{sb}] = \rho \sqsubseteq \mathcal{R}\llbracket\ b\ \rrbracket^{\mathsf{E}}}$. Now, by typing, for each $b' \in b_0,\ \overline{b}$ either $b' \in \Delta$ or $b' = w$. But by assumption we know that E is sound, so

$$\Gamma_E \vdash \mathcal{C}\llbracket\ b'\ \rrbracket^{\mathsf{E}}\ : \rho \sqsubseteq \mathcal{R}\llbracket\ b'\ \rrbracket^{\mathsf{E}}$$

for any $b' \in \Delta$. For the case $b' = w$ we have $\mathcal{R}[\![b']\!]^{\mathsf{E}} = \rho$ and $\mathcal{C}[\![b']\!]^{\mathsf{E}} = \mathbb{0}$, so rule Reflexive can be applied to yield the desired result.

case Do

Given $\Gamma \mid \Delta \vdash \textbf{do}\ c(v) : \tau_2$, we have $\Gamma \mid \Delta \vdash c : \textbf{Cap}\ \tau_1\ \tau_2$ and $\Gamma \vdash v : \tau_1$ for some τ_1. The assertion for blocks gives us $\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \vdash c : \textbf{Cap}\ \mathcal{R}[\![c]\!]^{\mathsf{E}}\ \tau_1\ \tau_2$ and the assertion for values yields $\Gamma_E,\ \Gamma \vdash v : \tau_1$. Since necessarily $c \in \Delta$, the soundness of E gives us

$$\Gamma_E \vdash \mathcal{C}[\![c]\!]^{\mathsf{E}} : \rho \sqsubseteq \mathcal{R}[\![c]\!]^{\mathsf{E}}$$

Hence, we can apply rule Do to obtain

$$\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \mid \rho \vdash \textbf{do}\ c[\ \mathcal{C}[\![c]\!]^{\mathsf{E}}\ ](v) : \tau_2$$

case Try

Given $\Gamma \mid \Delta \vdash \textbf{try}\ \{\ (c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \} : \tau$, we have $\Gamma \mid \Delta,\ c : \textbf{Cap}\ \tau_1\ \tau_2 \vdash s_0 : \tau$ and $\Gamma,\ x : \tau_1 \mid \Delta,\ k : \textbf{Cap}\ \tau_2\ \tau \vdash s : \tau$ for some $\tau_1,\ \tau_2$. We need to show that

$$\begin{aligned}&\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\Delta]\!]^{\mathsf{E}} \mid \rho \vdash \\ &\textbf{try}\ \{\ [r;\ n : r \sqsubseteq \rho](c) \Rightarrow \mathcal{S}[\![s_0]\!]^{\mathsf{E}'}\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow \mathcal{S}[\![s]\!]^{\mathsf{E}''}\ \} : \tau\end{aligned}$$

where $r,\ n$ are fresh and

$$\begin{aligned}\mathsf{E}' &:= \mathsf{E} \oplus n,\ c \mapsto (r \mid \mathbb{0}) \\ &= \{\ r,\ (\Delta,\ c),\ (m \oplus n,\ c \mapsto (r \mid \mathbb{0})),\ (\Gamma_E,\ r,\ n : r \sqsubseteq \rho)\ \} \\ \mathsf{E}'' &:= \mathsf{E},\ k \mapsto (\rho \mid \mathbb{0}) \\ &= \{\ \rho,\ (\Delta,\ k),\ (m,\ k \mapsto (\rho \mid \mathbb{0})),\ \Gamma_E\ \}\end{aligned}$$

To be able to invoke rule Try we thus need

$$\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\Delta]\!]^{\mathsf{E}},\ r,\ n : r \sqsubseteq \rho,\ c : \textbf{Cap}\ r\ \tau_1\ \tau_2 \mid r \vdash \mathcal{S}[\![s_0]\!]^{\mathsf{E}'} : \tau$$

and

$$\Gamma_E,\ \Gamma,\ \mathcal{T}[\![\Delta]\!]^{\mathsf{E}},\ x : \tau_1,\ k : \textbf{Cap}\ \rho\ \tau_2\ \tau \mid \rho \vdash \mathcal{S}[\![s]\!]^{\mathsf{E}''} : \tau$$

The induction hypothesis instantiated with E' (which is applicable since E' is sound by Lemma 5 and $\mathsf{E}'.\Delta = \Delta,\ c$) gives us

$$\Gamma_E,\ r,\ n : r \sqsubseteq \rho,\ \Gamma,\ \mathcal{T}[\![\Delta]\!]^{\mathsf{E}'},\ c : \textbf{Cap}\ \mathcal{R}[\![c]\!]^{\mathsf{E}'}\ \tau_1\ \tau_2 \mid r \vdash \mathcal{S}[\![s_0]\!]^{\mathsf{E}'} : \tau$$

and instantiated with E'' (which is applicable since E'' is sound by Lemma 5 and $\mathsf{E}''.\Delta = \Delta,\ k$) we obtain

$$\Gamma_E,\ \Gamma,\ x : \tau_1,\ \mathcal{T}[\![\Delta]\!]^{\mathsf{E}''},\ k : \textbf{Cap}\ \mathcal{R}[\![k]\!]^{\mathsf{E}''}\ \tau_2\ \tau \mid \rho \vdash \mathcal{S}[\![s]\!]^{\mathsf{E}''} : \tau$$

But note that we have

$$\begin{aligned}\mathcal{R}[\![g]\!]^{\mathsf{E}'} = \mathsf{pr}_1((m \oplus n)(g)) = \mathsf{pr}_1(m(g)) &= \mathcal{R}[\![g]\!]^{\mathsf{E}} \\ &= \mathcal{R}[\![g]\!]^{\mathsf{E}''}\end{aligned}$$

for all $g \in \Delta$ and hence $\mathcal{T}[\![\Delta]\!]^{\mathsf{E}'} = \mathcal{T}[\![\Delta]\!]^{\mathsf{E}''} = \mathcal{T}[\![\Delta]\!]^{\mathsf{E}}$. Moreover, we know that $\mathcal{R}[\![c]\!]^{\mathsf{E}'} = r$ and $\mathcal{R}[\![k]\!]^{\mathsf{E}''} = \rho$. Thus, we have what we need.

◀

A.2 Semantics Preservation

Let us now sktech a proof of the evaluation theorem for the lift-inference translation (Theorem 7).

▶ **Theorem 10** (Evaluation for the Lift-Inference Translation)**.**
If $\emptyset \mid \emptyset \vdash s : \tau$ *and* $\langle\, s \parallel \#_{start} :: \bullet \,\rangle \rightarrow^{n\,+\,k} \langle\, \textbf{return}\ v \parallel \bullet \,\rangle$,
then $\langle\, \mathcal{S}[\![\, s \,]\!]^{\emptyset} \parallel \#_{start} :: \bullet \,\rangle \rightarrow^{n\,+\,2k} \langle\, \textbf{return}\ v \parallel \bullet \,\rangle$,
where k *is the number of steps by rule* def *and* n *the number of other steps in* System Ξ.

Proof Sketch. We give a sketch how a proof may proceed.

1. We define a calculus $\Lambda^{\mathsf{Er}}_{\mathsf{cap}}$ to be the same as Λ_{cap} but with all region and evidence abstractions and applications omitted. We further define an erasure translation Er from Λ_{cap} to $\Lambda^{\mathsf{Er}}_{\mathsf{cap}}$ which erases all regions and evidence.
2. As the regions and evidence are immaterial for the operational semantics in Λ_{cap}, the machines of Λ_{cap} and $\Lambda^{\mathsf{Er}}_{\mathsf{cap}}$ proceed in lockstep. In particular, we have the following result: For M_1, if $\mathsf{Er}(\mathsf{M}_1) \rightarrow \mathsf{M}'_2$, there exists M_2 such that $\mathsf{M}_1 \rightarrow \mathsf{M}_2$ and $\mathsf{Er}(\mathsf{M}'_2) = \mathsf{M}_2$.
3. As a corollary we obtain the following evaluation result:
   If $\mathsf{Er}(\langle\, s \parallel \#_{\mathsf{start}} :: \bullet \,\rangle) \rightarrow^{n} \langle\, \textbf{return}\ v \parallel \bullet \,\rangle$, then $\langle\, s \parallel \#_{\mathsf{start}} :: \bullet \,\rangle \rightarrow^{n} \langle\, \textbf{return}\ v \parallel \bullet \,\rangle$.
   This follows from (1) by induction on n, using that $\mathsf{Er}(s') = \textbf{return}\ v$ implies $s' = \textbf{return}\ v$.
4. We can now compose the lift-inference translation with erasure to obtain a translation $[\![\, \cdot \,]\!]^{\mathsf{Er}} = \mathsf{Er} \circ [\![\, \cdot \,]\!]^{\emptyset}$ from System Ξ to $\Lambda^{\mathsf{Er}}_{\mathsf{cap}}$. Here we have used the empty lifting environment, but as the regions and evidence are erased anyway, this does not matter.
5. Now, since the lift-inference translation only adds regions and evidence without altering the constructs and structure otherwise and since the machine reductions in System Ξ are identical to those in Λ_{cap} modulo the regions and evidence (and rule *def*), we obtain the following simulation result for the composed translation:
   If $\mathsf{M}_1 \rightarrow \mathsf{M}_2$, then $[\![\, \mathsf{M}_1 \,]\!]^{\mathsf{Er}} \rightarrow^{1,\ 2} [\![\, \mathsf{M}_2 \,]\!]^{\mathsf{Er}}$, where 2 steps are needed if the original step is by rule *def* in which case the translated machine uses rule *push* immediately followed by rule *pop*.
6. As a corollary we obtain the following evaluation result for the composed translation:
   If $\emptyset \mid \emptyset \vdash s : \tau$ and $\langle\, s \parallel \#_{\mathsf{start}} :: \bullet \,\rangle \rightarrow^{n\,+\,k} \langle\, \textbf{return}\ v \parallel \bullet \,\rangle$,
   then $[\![\, \langle\, s \parallel \#_{\mathsf{start}} :: \bullet \,\rangle \,]\!]^{\mathsf{Er}} \rightarrow^{n\,+\,2k} \langle\, \textbf{return}\ v \parallel \bullet \,\rangle$,
   where k is the number of steps by rule *def* and n the number of other steps in System Ξ.
7. Now we can show the actual statement. Under the given assumptions, (4) and (6) yield $\mathsf{Er}(\langle\, \mathcal{S}[\![\, s \,]\!]^{\emptyset} \parallel \#_{\mathsf{start}} :: \bullet \,\rangle) \rightarrow^{n\,+\,2k} \langle\, \textbf{return}\ v \parallel \bullet \,\rangle$. Hence, we can conclude the desired result from (3).

◀

Syntax of Labels:

$l \quad ::= \quad$ `@a5f` $\mid$ `@4b2` $\mid \ldots$

Additional Runtime Blocks:

$b \quad ::= \quad \ldots \mid \textbf{cap}_l \; \{\, (x,\; k) \Rightarrow s \,\} \mid \textbf{cap}_l \; \mathsf{H} \mid \textbf{resume}(\mathsf{H})$

Syntax of Machine States:

$\mathsf{M} \quad ::= \quad \langle\, s \parallel \mathsf{K} \,\rangle$ executing

$\quad\quad \mid \quad \langle\, \textbf{do}\ b(v) \parallel \mathsf{K} \parallel \mathsf{H} \,\rangle$ unwinding

Syntax of Frames:

$\mathsf{F} \quad ::= \quad \textbf{val}\ x \ = \ \square;\ s$

$\quad\quad \mid \quad \textbf{resume}(\mathsf{H})\ \square$

Syntax of Meta Stacks:

$\mathsf{K} \quad ::= \quad \bullet \mid \mathsf{S} :: \ \mathsf{K}$

Syntax of Resumptions:

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

Syntax of Stacks:

$\mathsf{S} \quad ::= \quad \#_l \mid \mathsf{F} :: \ \mathsf{S}$

Syntactic Sugar:

$\mathsf{S}_l \ \doteq \ \overline{\mathsf{F} ::}\ \#_l$

Figure 10 Syntax of the abstract machine for System Ξ and Λ_{cap}.

B CPS Translation from Λ_{cap} to System F

In this appendix, we give some more details on the operational semantics of Λ_{cap} and the CPS translation from Λ_{cap} to System F.

B.1 Operational Semantics of Λ_{cap}

We first take a closer look at the machine semantics of Λ_{cap}. We define the operational semantics of the two calculi in terms of an abstract machine. The operational semantics of both languages is very similar. Therefore, we only show the machine for Λ_{cap} here and just describe the few differences for System Ξ.

There are only two minor differences. First, there is an additional stepping rule for function definition. Second, there are regions and evidence. However, both are irrelevant for the machine semantics as it proceeds by searching delimiters with labels on the meta stack and does not use region and evidence information. In the original version Brachthäuser et al. [4] define the operational semantics for System Ξ using evaluation contexts. It is not difficult to give an abstract machine formulation for their version (see *e.g.* [5] for how this is done for an extension of System Ξ). As our version treats continuations as capabilities, we deviate a bit from that abstract machine in how the handling of effect operations proceeds.

Syntax Figure 10 shows the syntax for our abstract machine. The machine uses labels l freshly generated at runtime to implement multi-prompt delimited control [9]. During execution the machine makes use of three additional runtime constructs. There are two kinds of runtime capabilities $\textbf{cap}_l \; \{ \ldots \}$ and $\textbf{cap}_l \; \mathsf{H}$, the former for handlers of effect operations and the latter for continuations. Both of them carry a label. The one for effect operations contains a handler implementation while the one for continuations contains a resumption. The third construct $\textbf{resume}(\mathsf{H})$ for resuming continuations also contains a resumption but no label.

The machine has two different kinds of states. The executing state $\langle\, s \parallel \mathsf{K} \,\rangle$ contains a statement s in focus to be evaluated and a runtime meta stack K which is a list of delimited stacks S. The unwinding state consists of a statement performing a capability, the meta stack K and a resumption H. This state is for unwinding the meta stack and pushing stacks onto the resumption. A resumption is thus also a list of stacks but in reverse order compared to the

Extended Syntax of Evidence and Regions.:

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

Evidence Values and Runtime Regions:

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

Runtime Region of Meta Stack:

$$
\begin{array}{lll}
\mathcal{R}[\![\ \cdot\ ]\!] & : & \mathsf{K} \to u \\
\mathcal{R}[\![\ \bullet\ ]\!] & = & \bullet \\
\mathcal{R}[\![\ \mathsf{S}_l :: \mathsf{K}\ ]\!] & = & l :: \mathcal{R}[\![\ \mathsf{K}\ ]\!]
\end{array}
$$

Normalization of Evidence:

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

Figure 11 Runtime regions and evidence.

meta stack. A stack is a list of frames ending with a delimiter $\#_l$ containing a label l. This is a minor technical difference to the original version of the language which treats delimiters as regular frames and does not explicitly segment the meta stack into delimited stacks. Frames are either sequencing frames **val** $x = \square;\ s$ or resumption frames **resume**(H) $\square$. To ease the presentation of the machine reduction steps a bit, we add some syntactic sugar for stacks: we write S_l for a stack whose delimiter has label l.

Figure 11 shows the part of the runtime syntax specific for Λ_{cap} and the calculation of runtime regions and evidence. Runtime regions and evidence both are lists of labels. The runtime region of a meta stack is the list of all labels in the delimiters present on the meta stack. Evidence is normalized to lists by using the empty list for trivial evidence and list concatenation for evidence composition.

Typing Figure 12 shows the typing rules for the abstract machine. The corresponding rules for System Ξ are almost identical, only the regions and evidence are dropped and the typing environments have to be split into separate value and block environments.

The typing of resuming continuations and of continuation capabilities is essentially the same, the only difference being that the label for the continuation capability is the topmost label of the runtime region for the capability. This is also the case for handler capabilities, but note how the region of the bound continuation variable does not contain this label. The premises for typing handler capabilities are the same as those for the implementation statement of a handler. Moreover, rule Evidence shows that runtime evidence is just the prefix constituting the difference of the lists representing the runtime regions.

Typing of executing machine states shows that the type of the statement in focus must be the type the meta stack expects. For unwinding states, the type of the statement must be the input type of the resumption and the output type of the resumption must be the type the meta stack expects. Moreover, the correct region is always given by the runtime region of the meta stack.

For the typing of stacks, meta stacks and resumptions note that the region for a stack

Runtime Value Typing $\boxed{\Gamma \vdash e : \gamma}$ $\boxed{\Gamma \vdash v : \tau}$

$$\frac{u_0 = w \mathbin{+\!\!+} u_1}{\emptyset \vdash w : u_0 \sqsubseteq u_1}\,[\textsc{Evidence}] \qquad \frac{\tau_1 \mid \rho \vdash \mathsf{H} : \tau_2}{\emptyset \vdash \textbf{resume}(\mathsf{H}) : \tau_1 \rightarrow\rho\ \tau_2}\,[\textsc{Continuation}]$$

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

$$\frac{\tau_1 \mid l :: \ \rho \vdash \mathsf{H} : \tau_2}{\emptyset \vdash \textbf{cap}_l\ \mathsf{H} : \textbf{Cap}\ (l :: \ \rho)\ \tau_1\ \tau_2}\,[\textsc{CapabilityK}]$$

Abstract Machine Typing $\boxed{\vdash \mathsf{M}\ \mathit{ok}}$

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

$$\frac{\emptyset \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash \textbf{do}\ v_0[w](v) : \tau_2 \qquad \vdash \mathsf{K} : \tau \qquad \tau_2 \mid \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash \mathsf{H} : \tau}{\vdash \langle\ \textbf{do}\ v_0[w](v) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle\ \mathit{ok}}\,[\textsc{Unwinding}]$$

Meta Stack Typing $\boxed{\vdash \mathsf{K} : \tau}$

$$\frac{}{\vdash \bullet : \tau}\,[\textsc{Exit}] \qquad \frac{\tau \mid l :: \ \mathcal{R}[\![\ \mathsf{K}\ ]\!] \vdash \mathsf{S}_l : \tau_0 \qquad \vdash \mathsf{K} : \tau_0}{\vdash \mathsf{S}_l :: \ \mathsf{K} : \tau}\,[\textsc{Stack}]$$

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

$$\frac{}{\tau \mid \rho \vdash \bullet : \tau}\,[\textsc{Exit}] \qquad \frac{\tau_1 \mid l :: \ \rho \vdash \mathsf{S}_l : \tau_2 \qquad \tau \mid l :: \ \rho \vdash \mathsf{H} : \tau_1}{\tau \mid \rho \vdash \mathsf{S}_l :: \ \mathsf{H} : \tau_2}\,[\textsc{Stack}]$$

Stack Typing $\boxed{\tau \mid \rho \vdash \mathsf{S} : \tau}$

$$\frac{}{\tau \mid l :: \ \rho \vdash \#_l : \tau}\,[\textsc{Handler}]$$

$$\frac{x : \tau_1 \mid \rho \vdash s : \tau_0 \qquad \tau_0 \mid \rho \vdash \mathsf{S} : \tau_2}{\tau_1 \mid \rho \vdash \textbf{val}\ x = \Box;\ s :: \ \mathsf{S} : \tau_2}\,[\textsc{Frame}] \qquad \frac{\tau_1 \mid \rho \vdash \mathsf{H} : \tau_0 \qquad \tau_0 \mid \rho \vdash \mathsf{S} : \tau_2}{\tau_1 \mid \rho \vdash \textbf{resume}(\mathsf{H})\ \Box :: \ \mathsf{S} : \tau_2}\,[\textsc{Resumption}]$$

Figure 12 Typing of the abstract machine for Λ_{cap}.

delimited by label l always contains l as the topmost label. All frames of the stack are then typed in that region. Each stack has an input type, which is the type of its hole, and an output type, which is the overall type of its lowermost frame. In the typing of meta stacks each stack must be typed in the region consisting of its label concatenated with the region of the remaining stack. The output type of the stack must be the type the meta stack expects. Resumptions are typed similarly, but since they contain stacks in reverse order the output type is the output type of the topmost stack and the input type is the input type of the lowermost stack. The region of the topmost stack and of the remaining resumption must agree, with the label of the stack being the topmost label in the region. The rest of the region stands for the region of the meta stack on top of which the resumption will run. This is because the topmost stack will be the first to be pushed back onto the meta stack when the resumption is actually used later. The whole resumption then can be typed without this topmost label, as it is present in the resumption itself and thus does not have to be present on the meta stack on top of which the resumption will run.

Machine Reductions:

Standard Machine Reductions

$$\begin{array}{lll}
\textit{(app)} & \langle\, \{\ [\overline{r}\ ;\ \overline{n:\ \gamma}](\overline{x:\ \tau})\ \textbf{at}\ \rho \Rightarrow s\ \}[\overline{\rho},\overline{e}](\overline{v}) \parallel \mathsf{K}\,\rangle & \to\ \langle\, s[\overline{r \mapsto \rho}, \overline{n \mapsto e}, \overline{x \mapsto v}] \parallel \mathsf{K}\,\rangle \\
\textit{(push)} & \langle\, \textbf{val}\ x\ =\ s_0;\ s \parallel \mathsf{S} ::\ \mathsf{K}\,\rangle & \to\ \langle\, s_0 \parallel (\textbf{val}\ x\ =\ \Box;\ s ::\ \mathsf{S}) ::\ \mathsf{K}\,\rangle \\
\textit{(pop)} & \langle\, \textbf{return}\ v \parallel (\textbf{val}\ x\ =\ \Box;\ s ::\ \mathsf{S}) ::\ \mathsf{K}\,\rangle & \to\ \langle\, s[x \mapsto v] \parallel \mathsf{S} ::\ \mathsf{K}\,\rangle \\
\textit{(res)} & \langle\, \textbf{return}\ v \parallel (\textbf{resume}(\mathsf{H})\ \Box ::\ \mathsf{S}) ::\ \mathsf{K}\,\rangle & \to\ \langle\, \textbf{resume}(\mathsf{H})(v) \parallel \mathsf{S} ::\ \mathsf{K}\,\rangle \\
\textit{(ret)} & \langle\, \textbf{return}\ v \parallel \#_l ::\ \mathsf{K}\,\rangle & \to\ \langle\, \textbf{return}\ v \parallel \mathsf{K}\,\rangle
\end{array}$$

Installing Effect Handlers

$$\begin{array}{ll}
\textit{(try)} & \langle\, \textbf{try}\ \{\ [r\ ;\ n](c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \} \parallel \mathsf{K}\,\rangle\ \to\ \langle\, s_0[r \mapsto u, n \mapsto w, c \mapsto v] \parallel \#_l ::\ \mathsf{K}\,\rangle \\
& \quad \text{where } l\ =\ \texttt{generateFresh}() \text{ and } v\ =\ \textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \} \\
& \quad \text{and } u\ =\ l :: \mathcal{R}[\![\ \mathsf{K}\ ]\!] \text{ and } w\ =\ l :: \bullet
\end{array}$$

Handling Effect Operations

$$\begin{array}{lll}
\textit{(perform)} & \langle\, \textbf{do}\ (\textbf{cap}_l\ h)[e](v) \parallel \mathsf{K}\,\rangle & \to\ \langle\, \textbf{do}\ (\textbf{cap}_l\ h)[\ N[\![e]\!]\ ](v) \parallel \mathsf{K} \parallel\ \bullet\ \rangle \\
\textit{(unwind)} & \langle\, \textbf{do}\ (\textbf{cap}_l\ h)[l' ::\ w](v) \parallel \mathsf{S}_{l'} ::\ \mathsf{K} \parallel \mathsf{H}\,\rangle & \to \\
& \quad \langle\, \textbf{do}\ (\textbf{cap}_l\ h)[w](v) \parallel \mathsf{K} \parallel \mathsf{S}_{l'} ::\ \mathsf{H}\,\rangle \quad \text{where } l \neq l' & \\
\textit{(handle)} & \langle\, \textbf{do}\ (\textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})[\bullet](v) \parallel \mathsf{S}_l ::\ \mathsf{S}'_{l'} ::\ \mathsf{K} \parallel \mathsf{H}\,\rangle & \to\ \langle\, s[x \mapsto v,\ k \mapsto j] \parallel \mathsf{S}'_{l'} ::\ \mathsf{K}\,\rangle \\
& \quad \text{where } j\ =\ \textbf{cap}_{l'}\ (\mathsf{S}_l ::\ \mathsf{H}) & \\
\textit{(rewindK)} & \langle\, \textbf{do}\ (\textbf{cap}_l\ \mathsf{H}')[\bullet](v) \parallel \mathsf{S}_l ::\ \mathsf{K} \parallel \mathsf{S} ::\ \mathsf{H}\,\rangle & \to \\
& \quad \langle\, \textbf{do}\ (\textbf{cap}_l\ \mathsf{H}')[\bullet](v) \parallel (\textbf{resume}(\mathsf{S} ::\ \mathsf{H})\ \Box ::\ \mathsf{S}_l) ::\ \mathsf{K} \parallel\ \bullet\ \rangle & \\
\textit{(handleK)} & \langle\, \textbf{do}\ (\textbf{cap}_l\ \mathsf{H})[\bullet](v) \parallel \mathsf{S}_l ::\ \mathsf{K} \parallel\ \bullet\ \rangle & \to\ \langle\, \textbf{resume}(\mathsf{H})(v) \parallel \mathsf{S}_l ::\ \mathsf{K}\,\rangle \\
\textit{(rewind)} & \langle\, \textbf{resume}(\mathsf{S} ::\ \mathsf{S}' ::\ \mathsf{H})(v) \parallel \mathsf{K}\,\rangle & \to\ \langle\, \textbf{resume}(\mathsf{S}' ::\ \mathsf{H})(v) \parallel \mathsf{S} ::\ \mathsf{K}\,\rangle \\
\textit{(resume-1)} & \langle\, \textbf{resume}((\textbf{val}\ x\ =\ \Box;\ s ::\ \mathsf{S}) ::\ \bullet)(v) \parallel \mathsf{K}\,\rangle & \to\ \langle\, s\ [x \mapsto v] \parallel \mathsf{S} ::\ \mathsf{K}\,\rangle \\
\textit{(resume-2)} & \langle\, \textbf{resume}((\textbf{resume}(\mathsf{H})\ \Box ::\ \mathsf{S}) ::\ \bullet)(v) \parallel \mathsf{K}\,\rangle & \to\ \langle\, \textbf{resume}(\mathsf{H})(v) \parallel \mathsf{S} ::\ \mathsf{K}\,\rangle \\
\textit{(resume-3)} & \langle\, \textbf{resume}(\#_l ::\ \bullet)(v) \parallel \mathsf{K}\,\rangle & \to\ \langle\, \textbf{return}\ v \parallel \mathsf{K}\,\rangle
\end{array}$$

Figure 13 Steps of the abstract machine for Λ_{cap}.

Reduction steps The reduction steps for the machine are given in Figure 13. Execution of a closed statement s always starts with a delimiter with a special toplevel label on the otherwise empty meta stack, that is, in state $\langle\, s \parallel \#_{\mathsf{start}} ::\ \bullet\,\rangle$.

Many of the rules are standard. The additional rule *def* in System Ξ for function definitions reads

$$\textit{(def)} \qquad \langle\, \textbf{def}\ f\ =\ w;\ \ s \parallel \mathsf{K}\,\rangle\ \to\ \langle\, s[f \mapsto w] \parallel \mathsf{K}\,\rangle$$

The newly defined block is substituted in the remaining statement. Similarly, in rule *app* the arguments of the function block are substituted in its body. Rule *push* focuses on the first statement and pushes a sequencing frame onto the topmost stack. When returning a value to such a frame (rule *pop*), execution goes on with the statement in that frame. Similarly, when returning to a resumption frame (rule *res*), execution continues with that resumption block. Returning to a stack that only consists of a delimiter at the end (rule *ret*) simply pops that delimiter and execution goes on with returning to the next stack. Upon execution of a handler statement (rule *try*) a fresh label is generated and a new stack consisting only of a delimiter with that label is pushed onto the meta stack. Execution then continues with the handled statement where the abstracted capability variable is replaced by a runtime handler capability with the just generated label l and the handler implementation.

When encountering a call to a capability, rule *perform* transitions to unwinding mode. Rule *unwind* then pops stacks off the meta stack and pushes them onto the resumption until the stack S_l ending with the delimiter with the correct label is found. Note how the labels in the evidence for capabilities precisely match the labels on the meta stack when unwinding. The rules so far are essentially the same as for the original version of Λ_{cap}. Now, in the original version rule *handle* would proceed by substituting the block $\textbf{resume}(\mathsf{S}_l :: \mathsf{H})$ for the continuation variable k in the handler statement s. However, we treat continuations as capabilities, so we instead substitute $\textbf{cap}_{l'}\ (\mathsf{S}_l :: \mathsf{H})$ where l' is the label of the delimiter of the next stack. This is the reason why we start execution with a toplevel delimiter, so that there always is a next stack. When encountering such a continuation capability, the machine again goes into unwinding mode as described before, but when the stack with the correct label is found, the collected resumption is pushed as a resumption frame onto that stack (rule *rewindK*). Rule *handleK* then transitions back to execution mode turning the continuation capability into a resumption block as its label is not needed anymore.

At this point the machine is in almost the same state as it would be in the original version. The only difference is that the part of the meta stack which in the original version would be on top of the stack S_l, where l is the correct label for the continuation, now is packaged into a resumption frame at the beginning of S_l. Such a resumption frame acts a bit like an "underflow" frame [10] when returning to it, in the sense that execution then first continues with that resumption (see rule *res*). When unwinding, however, it is treated just as another ordinary frame. Hence, when a call to a capability inside the continuation is encountered, it only sees the labels present on the meta stack when the continuation was created so that its evidence is correct. This difference in how continuations are treated compared to the original version of Λ_{cap} hence does not impact the final result of the execution for all programs that could be written in the original version.

Reduction of a resumption block proceeds by first rewinding the stacks in the resumption onto the meta stack (rule *rewind*), and then returning to the last stack in the resumption (with cutting one return step short, see rules *resume-1*, *resume-2*, *resume-3*).

B.2 CPS Translation

Figure 14 shows the CPS translation from Λ_{cap} to $\mathsf{System\ F}$ for the syntax and typing. It is defined over typing derivations but we abbreviate them by only writing the term. As noted before, the only difference compared to the original version of Λ_{cap} is that we distinguish handler and continuation capabilities and η-expand in the translation of the latter.

Translation of Types:

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

Translation of Values:

$$\begin{aligned}
\mathcal{V}\llbracket\ x\ \rrbracket &= x\\
\mathcal{V}\llbracket\ 1\ \rrbracket &= 1\\
\mathcal{V}\llbracket\ \{\ [\overline{r}, \overline{n\ :\ \gamma}](\overline{x\ :\ \tau})\ \textbf{at}\ \rho \Rightarrow s\ \}\ \rrbracket &= \overline{\Lambda r.}\ \overline{\lambda n.}\ \overline{\lambda x.}\ \mathcal{S}\llbracket\ s\ \rrbracket
\end{aligned}$$

Translation of Evidence:

$$\begin{aligned}
\mathcal{E}\llbracket\ n\ \rrbracket &= n\\
\mathcal{E}\llbracket\ \mathbb{0}\ \rrbracket &= \Lambda a.\ \mathcal{E}'\llbracket\ \mathbb{0}\ \rrbracket_a\\
\mathcal{E}\llbracket\ e_1\ \oplus e_2\ \rrbracket &= \Lambda a.\ \mathcal{E}'\llbracket\ e_1\ \oplus e_2\ \rrbracket_a\\
\mathcal{E}'\llbracket\ \mathbb{0}\ \rrbracket_a &= \lambda m.\ m\\
\mathcal{E}'\llbracket\ e_1\ \oplus e_2\ \rrbracket_a &= \lambda m.\ \mathcal{E}\llbracket\ e_1\ \rrbracket\ a\ (\mathcal{E}\llbracket\ e_2\ \rrbracket\ a\ m)
\end{aligned}$$

Translation of Statements:

$$\begin{aligned}
\mathcal{S}\llbracket\ \textbf{return}\ v\ \rrbracket &= \lambda k.\ k\ (\mathcal{V}\llbracket\ v\ \rrbracket)\\
\mathcal{S}\llbracket\ \textbf{val}\ x\ =\ s_0;\ s\ \rrbracket &= \lambda k.\ \mathcal{S}\llbracket\ s_0\ \rrbracket\ (\lambda x.\ \mathcal{S}\llbracket\ s\ \rrbracket\ k)\\
\mathcal{S}\llbracket\ v[\overline{\rho}, \overline{e}](\overline{v})\ \rrbracket &= \mathcal{V}\llbracket\ v\ \rrbracket\quad \overline{\mathcal{T}\llbracket\ \rho\ \rrbracket}\quad \overline{\mathcal{E}\llbracket\ e\ \rrbracket}\quad \overline{\mathcal{V}\llbracket\ v\ \rrbracket}\\
\mathcal{S}\llbracket\ \textbf{do}\ v_0[e](v)\ \rrbracket &= \mathcal{E}\llbracket\ e\ \rrbracket\quad \mathcal{T}\llbracket\ \tau_2\ \rrbracket\quad (\mathcal{V}\llbracket\ v_0\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket)\\
\mathcal{S}\llbracket\ \textbf{do}\ k[e](v)\ \rrbracket &= \mathcal{E}\llbracket\ e\ \rrbracket\quad \mathcal{T}\llbracket\ \tau_2\ \rrbracket\quad (\lambda k_0.\ \mathcal{V}\llbracket\ k\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket\ k_0)\\
\mathcal{S}\llbracket\ \textbf{try}\ \{\ [r;\ n](c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}\ \rrbracket &= \textsc{Reset}\ ((\Lambda r.\ \lambda n.\ \lambda c.\ \mathcal{S}\llbracket\ s_0\ \rrbracket)\\
&\qquad (\textsc{Cps}\ \mathcal{T}\llbracket\ \rho\ \rrbracket\ \mathcal{T}\llbracket\ \tau\ \rrbracket)\quad (\textsc{Lift})\quad (\lambda x.\ \lambda k.\ \mathcal{S}\llbracket\ s\ \rrbracket))
\end{aligned}$$

Auxiliary Definitions:

$$\begin{aligned}
\textsc{Cps}\ R\ A &\ =\ (A \to R) \to R\\[4pt]
\textsc{Reset} &\ :\ \textsc{Cps}\ (\textsc{Cps}\ R\ A)\ A \to \textsc{Cps}\ R\ A\\
\textsc{Reset}\ m &\ =\ m\ (\lambda x.\ \lambda k.\ k\ x)\\[4pt]
\textsc{Lift} &\ :\ \forall a.\ \ \textsc{Cps}\ R\ a \to \textsc{Cps}\ (\textsc{Cps}\ R\ R')\ a\\
\textsc{Lift} &\ =\ \Lambda a.\ \lambda m.\ \lambda k.\ \lambda j.\ m\ (\lambda x.\ k\ x\ j)
\end{aligned}$$

Figure 14 CPS Translation from Λ_{cap} to System F.

Syntax of System F**:**

$$\begin{array}{lll}
\text{Terms} & & \\
t & ::= & x \mid \lambda x.\ t \mid \Lambda a.\ t \mid t\ t \mid t\ \tau \mid \texttt{done} \\
\text{Contexts} & & \\
A & ::= & \Box\ v \mid \textbf{let}\ \kappa\ =\ v\ \textbf{in}\ \Box \\
C & ::= & \bullet \mid A :: C
\end{array}$$

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

Translation of Machine States:

$$\begin{array}{lcl}
\mathcal{M}\llbracket\ \langle\ s \parallel \mathsf{K}\ \rangle\ \rrbracket & = & \textsc{Plug}(\mathcal{S}\llbracket\ s\ \rrbracket\ ,\ \Box\ \kappa :: \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \\
\mathcal{M}\llbracket\ \langle\ \textbf{do}\ \textbf{cap}_l\ \{\ (x, k) \Rightarrow s\ \}[w](v) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle\ \rrbracket & = & \\
\qquad \textsc{Plug}(\mathcal{W}\llbracket\ w\ \rrbracket_{(\lambda k.\ \mathcal{S}\llbracket\ s\ \rrbracket)[x\ \mapsto\ \mathcal{V}\llbracket\ v\ \rrbracket]},\ \Box\ \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket :: \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & & \\
\mathcal{M}\llbracket\ \langle\ \textbf{do}\ \textbf{cap}_l\ (\mathsf{S} :: \mathsf{H}')[w](v) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle\ \rrbracket & = & \\
\qquad \textsc{Plug}(\mathcal{W}\llbracket\ w\ \rrbracket_{(\lambda k.\ \mathcal{H}\llbracket\ \mathsf{H}'\ \rrbracket\ [\kappa\ \mapsto\ \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket]\ \mathcal{V}\llbracket\ v\ \rrbracket\ k)},\ \Box\ \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket :: \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & & \\
\mathcal{M}\llbracket\ \langle\ \textbf{do}\ \textbf{cap}_l\ (\bullet)[w](v) \parallel \mathsf{K} \parallel \mathsf{H}\ \rangle\ \rrbracket & = & \mathit{dummy}(\mathit{does\ not\ occur})
\end{array}$$

Translation of Meta Stacks:

$$\begin{array}{lcl}
\mathcal{K}\llbracket\ \bullet\ \rrbracket & = & \textbf{let}\ \kappa\ =\ \texttt{done}\ \textbf{in}\ \Box :: \bullet \\
\mathcal{K}\llbracket\ \mathsf{S} :: \mathsf{K}\ \rrbracket & = & \mathcal{F}\llbracket\ \mathsf{S}\ \rrbracket :: \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket
\end{array}$$

Translation of Resumptions:

$$\begin{array}{lcl}
\mathcal{H}\llbracket\ \bullet\ \rrbracket & = & \kappa \\
\mathcal{H}\llbracket\ \mathsf{S} :: \mathsf{H}\ \rrbracket & = & \lambda x_0.\ \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket]\ x_0\ \kappa
\end{array}$$

Translation of Stacks:

$$\begin{array}{lcl}
\mathcal{F}\llbracket\ \#_l\ \rrbracket & = & \textbf{let}\ \kappa\ =\ \lambda x.\ \lambda k.\ k\ x\ \textbf{in}\ \Box :: \Box\ \kappa \\
\mathcal{F}\llbracket\ \textbf{val}\ x\ =\ \Box;\ s :: \mathsf{S}\ \rrbracket & = & \textbf{let}\ \kappa\ =\ \lambda x.\ \mathcal{S}\llbracket\ s\ \rrbracket\ \kappa\ \textbf{in}\ \Box :: \mathcal{F}\llbracket\ \mathsf{S}\ \rrbracket \\
\mathcal{F}\llbracket\ \textbf{resume}(\mathsf{H})\ \Box :: \mathsf{S}\ \rrbracket & = & \textbf{let}\ \kappa\ =\ \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ \textbf{in}\ \Box :: \mathcal{F}\llbracket\ \mathsf{S}\ \rrbracket \\
& & \\
\mathcal{G}\llbracket\ \#_l\ \rrbracket & = & \lambda x.\ \lambda k.\ k\ x \\
\mathcal{G}\llbracket\ \textbf{val}\ x\ =\ \Box;\ s :: \mathsf{S}\ \rrbracket & = & \lambda x.\ \mathcal{S}\llbracket\ s\ \rrbracket\ \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket \\
\mathcal{G}\llbracket\ \textbf{resume}(\mathsf{H})\ \Box :: \mathsf{S}\ \rrbracket & = & \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket]
\end{array}$$

Translation of Unwinding:

$$\begin{array}{lcl}
\mathcal{W}\llbracket\ \bullet\ \rrbracket_t & = & t \\
\mathcal{W}\llbracket\ l :: w\ \rrbracket_t & = & \lambda k.\ \lambda j.\ \mathcal{W}\llbracket\ w\ \rrbracket_t\ (\lambda x.\ k\ x\ j)
\end{array}$$

Translation of Evidence Values:

$$\begin{array}{lcl}
\mathcal{E}\llbracket\ w\ \rrbracket & = & \Lambda a.\ \mathcal{E}'\llbracket\ w\ \rrbracket_a \\
\mathcal{E}'\llbracket\ w\ \rrbracket_a & = & \lambda m.\ \mathcal{W}\llbracket\ w\ \rrbracket_m
\end{array}$$

Translation of Runtime Regions:

$$\begin{array}{lcl}
\mathcal{T}\llbracket\ \bullet\ \rrbracket & = & \mathsf{Void} \\
\mathcal{T}\llbracket\ l :: \rho\ \rrbracket & = & \textsc{Cps}\ \mathcal{T}\llbracket\ \rho\ \rrbracket\ \mathcal{T}\llbracket\ \tau\ \rrbracket \qquad \text{where } l \text{ is delimited at } \tau
\end{array}$$

Translation of Runtime Values:

$$\begin{array}{lcl}
\mathcal{V}\llbracket\ \textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}\ \rrbracket & = & \lambda x.\ \lambda k.\ \mathcal{S}\llbracket\ s\ \rrbracket \\
\mathcal{V}\llbracket\ \textbf{resume}(\mathsf{S} :: \mathsf{H})\ \rrbracket & = & \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket] \\
\mathcal{V}\llbracket\ \textbf{resume}(\bullet)\ \rrbracket & = & \mathit{dummy}(\mathit{does\ not\ occur}) \\
\mathcal{V}\llbracket\ \textbf{cap}_l\ (\mathsf{S} :: \mathsf{H})\ \rrbracket & = & \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket] \\
\mathcal{V}\llbracket\ \textbf{cap}_l\ (\bullet)\ \rrbracket & = & \mathit{dummy}(\mathit{does\ not\ occur})
\end{array}$$

Figure 15 Translation of machine states.

Translation of runtime constructs Figure 15 shows the translation of the runtime constructs for the abstract machine. This is necessary to prove the simulation theorem. It is again similar to the original translation in [36] but differs in several regards. The syntax for System F and the auxiliary contexts and plugging are the same. In the translation of machine states, however, we have an additional case for continuation capabilities for the unwinding state. Here, the η-expansion mentioned above is again visible. Moreover, we use that the resumption can never be empty. The translation of stacks, meta stacks and resumptions is a bit different as meta stacks are explicitly partitioned, but the essence is still the same. Note that the translation of a resumption always contains κ free exactly once. The translation of stacks has two versions, one uses auxiliary contexts, the other one has the plugging done already. In the translation of runtime values there now additionally is a case for continuation capabilities. Note that its translation is exactly the same as for resuming continuations, because the label does not matter for the CPS translation.

B.3 Proofs for the CPS Translation

B.3.1 Semantics Preservation

Now we prove the simulation theorem for the CPS translation from Λ_{cap} to System F. We first state and prove several lemmas we need.

▶ **Lemma 11** (Subst).
$\mathcal{S}[\![\ s\ ]\!]\ [r \mapsto \mathcal{T}[\![\ \rho\ ]\!]\ ]\ =\ \mathcal{S}[\![\ s\ [r \mapsto \rho]\ ]\!]$
$\mathcal{S}[\![\ s\ ]\!]\ [n \mapsto \mathcal{E}[\![\ e\ ]\!]\ ]\ =\ \mathcal{S}[\![\ s\ [n \mapsto e]\ ]\!]$
$\mathcal{S}[\![\ s\ ]\!]\ [x \mapsto \mathcal{V}[\![\ v\ ]\!]\ ]\ =\ \mathcal{S}[\![\ s\ [x \mapsto v]\ ]\!]$

Proof. The only change compared to the original system is in the translation of the call of continuations. But that case is essentially the same as the one for calling a capability. Hence, the original proof carries over almost unchanged. ◀

▶ **Lemma 12** (Perform).
If e does not contain variables, then $\mathcal{E}'[\![\ e\ ]\!]_a\ v \to^* \mathcal{W}[\![\ N[\![\ e\ ]\!]\ ]\!]_v$

Proof. Still exactly the same as the original version. ◀

▶ **Lemma 13** (PlugRed).
If $t \to t'$ *then* $\textsc{Plug}(t,\ C) \to \textsc{Plug}(t',\ C)$

Proof. Still exactly the same as the original version. ◀

▶ **Lemma 14** (PlugSeg).
$\textsc{Plug}(t,\ \overline{A ::}\ \mathcal{F}[\![\ S\ ]\!] ::\ C)\ =\ \textsc{Plug}(t,\ \overline{A ::}\ \textbf{\textit{let}}\ \kappa\ =\ \mathcal{G}[\![\ S\ ]\!]\ \textbf{\textit{in}}\ \Box ::\ \Box\ \kappa ::\ C)$

Proof. By definition of $\textsc{Plug}$ there exists t' such that $\textsc{Plug}(t,\ \overline{A} ::\ C)\ =\ \textsc{Plug}(t',\ C)$. Thus, we can assume $\overline{A}$ to be empty without loss of generality. We prove by induction on S.

case $\mathsf{S}\ =\ \#_l$

$$\begin{array}{ll} \textsc{Plug}(t,\ \mathcal{F}[\![\ \#_l\ ]\!] ::\ C) & = \\ \textsc{Plug}(t,\ \textbf{let}\ \kappa\ =\ \lambda x.\ \lambda k.\ k\ x\ \textbf{in}\ \Box ::\ \Box\ \kappa ::\ C) & = \\ \textsc{Plug}(t,\ \textbf{let}\ \kappa\ =\ \mathcal{G}[\![\ \bullet\ ]\!]\ \textbf{in}\ \Box ::\ \Box\ \kappa ::\ C) & \end{array}$$

case $\mathsf{S}\ =\ \mathsf{F} ::\ \mathsf{S}'$

We distinguish cases for F.
For $\mathsf{F} \;=\; \textbf{val}\ x \;=\; \Box;\ s$ we have

$$\begin{array}{ll}
\textsc{Plug}(t,\ \mathcal{F}[\![\ \textbf{val}\ x \;=\; \Box;\ s :: \ \mathsf{S}'\ ]\!] :: \ C) & = \\
\textsc{Plug}(t,\ \textbf{let}\ \kappa \;=\; \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa\ \textbf{in}\ \Box :: \ \mathcal{F}[\![\ \mathsf{S}'\ ]\!] :: \ C) & = \\
\textsc{Plug}(t\ [\kappa \mapsto \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa],\ \mathcal{F}[\![\ \mathsf{S}'\ ]\!] :: \ C) & = \\
\textsc{Plug}(t\ [\kappa \mapsto \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa],\ \textbf{let}\ \kappa \;=\; \mathcal{G}[\![\ \mathsf{S}'\ ]\!]\ \textbf{in}\ \Box :: \ \Box\ \kappa :: \ C) & = \\
\textsc{Plug}(t\ [\kappa \mapsto \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \mathcal{G}[\![\ \mathsf{S}'\ ]\!]],\ \Box\ \kappa :: \ C) & = \\
\textsc{Plug}(t,\ \textbf{let}\ \kappa \;=\; \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \mathcal{G}[\![\ \mathsf{S}'\ ]\!]\ \textbf{in}\ \Box :: \ \Box\ \kappa :: \ C) & = \\
\textsc{Plug}(t,\ \textbf{let}\ \kappa \;=\; \mathcal{G}[\![\ \textbf{val}\ x \;=\; \Box;\ s :: \ \mathsf{S}'\ ]\!]\ \textbf{in}\ \Box :: \ \Box\ \kappa :: \ C) &
\end{array}$$

Here we have used that κ occurs free at most once on the left-hand side.
For $\mathsf{F} \;=\; \textbf{resume}(\mathsf{H})\ \Box$ we have

$$\begin{array}{ll}
\textsc{Plug}(t,\ \mathcal{F}[\![\ \textbf{resume}(\mathsf{H})\ \Box :: \ \mathsf{S}'\ ]\!] :: \ C) & = \\
\textsc{Plug}(t,\ \textbf{let}\ \kappa \;=\; \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ \textbf{in}\ \Box :: \ \mathcal{F}[\![\ \mathsf{S}'\ ]\!] :: \ C) & = \\
\textsc{Plug}(t\ [\kappa \mapsto \mathcal{H}[\![\ \mathsf{H}\ ]\!]],\ \mathcal{F}[\![\ \mathsf{S}'\ ]\!] :: \ C) & = \\
\textsc{Plug}(t\ [\kappa \mapsto \mathcal{H}[\![\ \mathsf{H}\ ]\!]],\ \textbf{let}\ \kappa \;=\; \mathcal{G}[\![\ \mathsf{S}'\ ]\!]\ \textbf{in}\ \Box :: \ \Box\ \kappa :: \ C) & = \\
\textsc{Plug}(t\ [\ \kappa \mapsto \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}'\ ]\!]]\ ],\ \Box\ \kappa :: \ C) & = \\
\textsc{Plug}(t,\ \textbf{let}\ \kappa \;=\; \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}'\ ]\!]]\ \textbf{in}\ \Box :: \ \Box\ \kappa :: \ C) & = \\
\textsc{Plug}(t,\ \textbf{let}\ \kappa \;=\; \mathcal{G}[\![\ \textbf{resume}(\mathsf{H})\ \Box :: \ \mathsf{S}'\ ]\!]\ \textbf{in}\ \Box :: \ \Box\ \kappa :: \ C) &
\end{array}$$

Here we have used that κ occurs free at most once on the left-hand side and does so in $\mathcal{H}[\![\ \mathsf{H}\ ]\!]$.

◀

▶ **Lemma 15** (ResumpSubst).
$\mathcal{H}[\![\ H\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ S\ ]\!]]$ always is a value and $\mathcal{G}[\![\ S\ ]\!]$ always is a value.

Proof. We prove both parts simultaneously. For the first part we use induction over H.

case $\mathsf{H} \;=\; \bullet$
We have $\mathcal{H}[\![\ \bullet\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]] \;=\; \mathcal{G}[\![\ \mathsf{S}\ ]\!]$, so the claim follows from the second part.

case $\mathsf{H} \;=\; \mathsf{S}' :: \ \mathsf{H}'$
We have $\mathcal{H}[\![\ \mathsf{S}' :: \ \mathsf{H}'\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]] \;=\; \lambda x_0.\ \mathcal{H}[\![\ \mathsf{H}'\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}'\ ]\!]]\ x_0\ \mathcal{G}[\![\ \mathsf{S}\ ]\!]$, which is a λ-abstraction and thus a value. Here we have used that κ does not occur free in $\mathcal{G}[\![\ \mathsf{S}'\ ]\!]$.

For the second part we use induction over S.

case $\mathsf{S} \;=\; \#_l$
We have $\mathcal{G}[\![\ \#_l\ ]\!] \;=\; \lambda x.\ \lambda k.\ k\ x$, which is a λ-abstraction and thus a value.

case $\mathsf{S} \;=\; \mathsf{F} :: \ \mathsf{S}'$
We distinguish cases for F.
For $\mathsf{F} \;=\; \textbf{val}\ x \;=\; \Box;\ s$, we have $\mathcal{G}[\![\ \textbf{val}\ x \;=\; \Box;\ s :: \ \mathsf{S}'\ ]\!] \;=\; \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \mathcal{G}[\![\ \mathsf{S}'\ ]\!]$, which is a λ-abstraction and thus a value.
For $\mathsf{F} \;=\; \textbf{resume}(\mathsf{H})\ \Box$, we have $\mathcal{G}[\![\ \textbf{resume}(\mathsf{H})\ \Box ::: \ \mathsf{S}'\ ]\!] \;=\; \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}'\ ]\!]]$, which is a value by the first part.

◀

▶ **Lemma 16** (Resume).

$$\begin{aligned}
&\mathcal{M}[\![\ \langle \ \textit{\textbf{resume}}(S :: \ H)(v) \parallel K \ \rangle \ ]\!] &&= \\
&\textsc{Plug}(\mathcal{H}[\![\ H \ ]\!] \ [\kappa \mapsto \mathcal{G}[\![\ S \ ]\!]] \ \mathcal{V}[\![\ v \ ]\!], \ \Box \ \kappa :: \ \mathcal{K}[\![\ K \ ]\!])
\end{aligned}$$

Proof.

$$\begin{aligned}
&\mathcal{M}[\![\ \langle \ \textbf{resume}(\mathsf{S} :: \ \mathsf{H})(v) \parallel \mathsf{K} \ \rangle \ ]\!] &&= \text{ by Def } \mathcal{M}[\![\ \cdot \ ]\!] \\
&\textsc{Plug}(\mathcal{S}[\![\ \textbf{resume}(\mathsf{S} :: \ \mathsf{H})(v) \ ]\!], \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!]) &&= \text{ by Def } \mathcal{S}[\![\ \cdot \ ]\!] \\
&\textsc{Plug}(\mathcal{V}[\![\ \textbf{resume}(\mathsf{S} :: \ \mathsf{H})]\!] \ \mathcal{V}[\![\ v \ ]\!], \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!]) &&= \text{ by Def } \mathcal{V}[\![\ \cdot \ ]\!] \\
&\textsc{Plug}(\mathcal{H}[\![\ \mathsf{H} \ ]\!] \ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S} \ ]\!]] \ \mathcal{V}[\![\ v \ ]\!], \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!])
\end{aligned}$$

◀

We are now ready to give a proof for the simulation theorem (Theorem 1).

▶ **Theorem 17** (Simulation for the CPS Translation).
If $\vdash \ M$ *ok and* $M \to M'$*, then* $\mathcal{M}[\![\ M \ ]\!] \to^* \mathcal{M}[\![\ M' \ ]\!]$*.*

Proof. The proof proceeds by case distinction on the stepping relation of the machine. We use Lemma PlugRed in most cases to reduce inside Plug. Moreover, we use that κ occurs free at most once on the left-hand side in Plug and thus is always substituted only in one place.

case *app*

$$\begin{aligned}
&\mathcal{M}[\![\ \langle \ \{ \ [\overline{r} \ ; \ \overline{n : \ \gamma}](\overline{x : \ \tau}) \ \textbf{at} \ \rho \Rightarrow s_0 \ \}[\overline{\rho}, \overline{e}](\overline{v}) \parallel \mathsf{K} \ \rangle \ ]\!] &&= \text{ by Def } \mathcal{M}[\![\ \cdot \ ]\!] \\
&\textsc{Plug}(\mathcal{S}[\![\ \{ \ [\overline{r} \ ; \ \overline{n : \ \gamma}](\overline{x : \ \tau}) \ \textbf{at} \ \rho \Rightarrow s_0 \ \}[\overline{\rho}, \overline{e}](\overline{v}) \ ]\!], \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!]) &&= \text{ by Def } \mathcal{S}[\![\ \cdot \ ]\!] \\
&\textsc{Plug}(\mathcal{V}[\![\ \{ \ [\overline{r} \ ; \ \overline{n : \ \gamma}](\overline{x : \ \tau}) \ \textbf{at} \ \rho \Rightarrow s_0 \ \} \ ]\!] \ \overline{\mathcal{T}[\![\ \rho \ ]\!]} \ \overline{\mathcal{E}[\![\ e \ ]\!]} \ \overline{\mathcal{V}[\![\ v \ ]\!]}, \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!]) &&= \text{ by Def } \mathcal{V}[\![\ \cdot \ ]\!] \\
&\textsc{Plug}((\overline{\Lambda r.} \ \overline{\lambda n.} \ \overline{\lambda x.} \ \mathcal{S}[\![\ s_0 \ ]\!]) \ \overline{\mathcal{T}[\![\ \rho \ ]\!]} \ \overline{\mathcal{E}[\![\ e \ ]\!]} \ \overline{\mathcal{V}[\![\ v \ ]\!]}, \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!]) &&\to^3 \text{ by } \beta\text{-reduction} \\
&\textsc{Plug}(\mathcal{S}[\![\ s_0 \ ]\!] \ [\overline{r \mapsto \mathcal{T}[\![\ \rho \ ]\!]} \ \overline{n \mapsto \mathcal{E}[\![\ e \ ]\!]} \ \overline{x \mapsto \mathcal{V}[\![\ v \ ]\!]}], \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!]) &&= \text{ by Lemma } \textsc{Subst} \\
&\textsc{Plug}(\mathcal{S}[\![\ s_0 \ [\overline{r \mapsto \rho} \ \overline{n \mapsto e} \ \overline{x \mapsto v}] \ ]\!], \ \Box \ \kappa :: \ \mathcal{K}[\![\ \mathsf{K} \ ]\!]) &&= \text{ by Def } \mathcal{M}[\![\ \cdot \ ]\!] \\
&\mathcal{M}[\![\ \langle \ s_0[\overline{r \mapsto \rho}, \overline{n \mapsto e}, \overline{x \mapsto v}] \parallel \mathsf{K} \ \rangle \ ]\!]
\end{aligned}$$

case *push*

$$\begin{array}{lll}
\mathcal{M}[\![\ \langle\ \textsf{\textbf{val}}\ x\ =\ s_0;\ s\ \|\ \mathsf{S} :: \ \mathsf{K}\ \rangle\ ]\!] & = & \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{S}[\![\ \textsf{\textbf{val}}\ x\ =\ s_0;\ s\ ]\!],\ \Box\ \kappa :: \ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \mathcal{S}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\lambda k.\ \mathcal{S}[\![\ s_0\ ]\!]\ (\lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ k),\ \Box\ \kappa :: \ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \textsc{Plug} \\
\textsc{Plug}((\lambda k.\ \mathcal{S}[\![\ s_0\ ]\!]\ (\lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ k))\ \kappa,\ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & \to & \text{by } \beta\text{-reduction} \\
\textsc{Plug}(\mathcal{S}[\![\ s_0\ ]\!]\ (\lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa),\ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \textsc{Plug} \\
\textsc{Plug}(\mathcal{S}[\![\ s_0\ ]\!],\ \Box\ \kappa :: \ \textsf{\textbf{let}}\ \kappa\ =\ \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa\ \textsf{\textbf{in}}\ \Box :: \ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Defs } \mathcal{K}[\![\ \cdot\ ]\!],\ \mathcal{F}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{S}[\![\ s_0\ ]\!],\ \Box\ \kappa :: \ \mathcal{K}[\![\ (\textsf{\textbf{val}}\ x\ =\ \Box;\ s :: \ \mathsf{S}) :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!] \\
\mathcal{M}[\![\ \langle\ s_0\ \|\ (\textsf{\textbf{val}}\ x\ =\ \Box;\ s :: \ \mathsf{S}) :: \ \mathsf{K}\ \rangle\ ]\!] & &
\end{array}$$

Here we have used that k is fresh.

case *pop*

$$\begin{array}{lll}
\mathcal{M}[\![\ \langle\ \textsf{\textbf{return}}\ v\ \|\ (\textsf{\textbf{val}}\ x\ =\ \Box;\ s :: \ \mathsf{S}) :: \ \mathsf{K}\ \rangle\ ]\!] & = & \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{S}[\![\ \textsf{\textbf{return}}\ v\ ]\!],\ \Box\ \kappa :: \ \mathcal{K}[\![\ (\textsf{\textbf{val}}\ x\ =\ \Box;\ s :: \ \mathsf{S}) :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \mathcal{S}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\lambda k.\ k\ \mathcal{V}[\![\ v\ ]\!],\ \Box\ \kappa :: \ \mathcal{K}[\![\ (\textsf{\textbf{val}}\ x\ =\ \Box;\ s :: \ \mathsf{S}) :: \ \mathsf{K}\ ]\!]) & = & \text{by Defs } \mathcal{K}[\![\ \cdot\ ]\!],\ \mathcal{F}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\lambda k.\ k\ \mathcal{V}[\![\ v\ ]\!],\ \Box\ \kappa :: \ \textsf{\textbf{let}}\ \kappa\ =\ \lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa\ \textsf{\textbf{in}}\ \Box :: \ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \textsc{Plug} \\
\textsc{Plug}((\lambda k.\ k\ \mathcal{V}[\![\ v\ ]\!])\ (\lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa),\ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & \to & \text{by } \beta\text{-reduction} \\
\textsc{Plug}((\lambda x.\ \mathcal{S}[\![\ s\ ]\!]\ \kappa)\ \mathcal{V}[\![\ v\ ]\!],\ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & \to & \text{by } \beta\text{-reduction} \\
\textsc{Plug}(\mathcal{S}[\![\ s\ ]\!]\ [x \mapsto \mathcal{V}[\![\ v\ ]\!]]\ \kappa,\ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \textsc{Plug} \\
\textsc{Plug}(\mathcal{S}[\![\ s\ ]\!]\ [x \mapsto \mathcal{V}[\![\ v\ ]\!]],\ \Box\ \kappa :: \ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Lemma } \textsc{Subst} \\
\textsc{Plug}(\mathcal{S}[\![\ s\ [x \mapsto v]\ ]\!],\ \Box\ \kappa :: \ \mathcal{K}[\![\ \mathsf{S} :: \ \mathsf{K}\ ]\!]) & = & \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!] \\
\mathcal{M}[\![\ \langle\ s[x \mapsto v]\ \|\ \mathsf{S} :: \ \mathsf{K}\ \rangle\ ]\!] & &
\end{array}$$

Here we have used that k is fresh and x only occurs free in s.

case *res*

$$\begin{array}{lll}
\mathcal{M}[\![\, \langle\, \textsf{\textbf{return}}\ v \parallel (\textsf{\textbf{resume}}(\mathsf{S}' :: \mathsf{H})\ \square :: \mathsf{S}) :: \mathsf{K} \,\rangle \,]\!] & = & \text{by Def } \mathcal{M}[\![\cdot]\!] \\
\textsc{Plug}(\mathcal{S}[\![\textsf{\textbf{return}}\ v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![(\textsf{\textbf{resume}}(\mathsf{S}' :: \mathsf{H})\ \square :: \mathsf{S}) :: \mathsf{K}]\!]) & = & \text{by Def } \mathcal{S}[\![\cdot]\!] \\
\textsc{Plug}(\lambda k.\ k\ \mathcal{V}[\![v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![(\textsf{\textbf{resume}}(\mathsf{S}' :: \mathsf{H})\ \square :: \mathsf{S}) :: \mathsf{K}]\!]) & = & \text{by Defs } \mathcal{K}[\![\cdot]\!],\ \mathcal{F}[\![\cdot]\!] \\
\textsc{Plug}(\lambda k.\ k\ \mathcal{V}[\![v]\!], & & \\
\qquad \square\ \kappa :: \textsf{\textbf{let}}\ \kappa\ =\ \mathcal{H}[\![\mathsf{S}' :: \mathsf{H}]\!]\ \textsf{\textbf{in}}\ \square ::\ \mathcal{K}[\![\mathsf{S} :: \mathsf{K}]\!]) & = & \text{by Def } \textsc{Plug} \\
\textsc{Plug}((\lambda k.\ k\ \mathcal{V}[\![v]\!])\ \mathcal{H}[\![\mathsf{S}' :: \mathsf{H}]\!],\ \mathcal{K}[\![\mathsf{S} :: \mathsf{K}]\!]) & \rightarrow & \text{by } \beta\text{-reduction} \\
\textsc{Plug}(\mathcal{H}[\![\mathsf{S}' :: \mathsf{H}]\!]\ \mathcal{V}[\![v]\!],\ \mathcal{K}[\![\mathsf{S} :: \mathsf{K}]\!]) & = & \text{by Def } \mathcal{H}[\![\cdot]\!] \\
\textsc{Plug}((\lambda x_0.\ \mathcal{H}[\![\mathsf{H}]\!]\ [\kappa \mapsto \mathcal{G}[\![\mathsf{S}']\!]]\ x_0\ \kappa)\ \mathcal{V}[\![v]\!],\ \mathcal{K}[\![\mathsf{S} :: \mathsf{K}]\!]) & \rightarrow & \text{by } \beta\text{-reduction} \\
\textsc{Plug}(\mathcal{H}[\![\mathsf{H}]\!]\ [\kappa \mapsto \mathcal{G}[\![\mathsf{S}']\!]]\ \mathcal{V}[\![v]\!]\ \kappa,\ \mathcal{K}[\![\mathsf{S} :: \mathsf{K}]\!]) & = & \text{by Def } \textsc{Plug} \\
\textsc{Plug}(\mathcal{H}[\![\mathsf{H}]\!]\ [\kappa \mapsto \mathcal{G}[\![\mathsf{S}']\!]]\ \mathcal{V}[\![v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![\mathsf{S} :: \mathsf{K}]\!]) & = & \text{by Lemma } \textsc{Resume} \\
\mathcal{M}[\![\, \langle\, \textsf{\textbf{resume}}(\mathsf{S}' :: \mathsf{H})(v) \parallel \mathsf{S} :: \mathsf{K} \,\rangle \,]\!] & &
\end{array}$$

Here we have used that x_0 is fresh.

case *ret*

$$\begin{array}{lll}
\mathcal{M}[\![\, \langle\, \textsf{\textbf{return}}\ v \parallel \#_l :: \mathsf{K} \,\rangle \,]\!] & = & \text{by Def } \mathcal{M}[\![\cdot]\!] \\
\textsc{Plug}(\mathcal{S}[\![\textsf{\textbf{return}}\ v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![\#_l :: \mathsf{K}]\!]) & = & \text{by Def } \mathcal{S}[\![\cdot]\!] \\
\textsc{Plug}(\lambda k.\ k\ \mathcal{V}[\![v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![\#_l :: \mathsf{K}]\!]) & = & \text{by Defs } \mathcal{K}[\![\cdot]\!],\ \mathcal{F}[\![\cdot]\!] \\
\textsc{Plug}(\lambda k.\ k\ \mathcal{V}[\![v]\!],\ \square\ \kappa :: \textsf{\textbf{let}}\ \kappa\ =\ \lambda x_1.\ \lambda k_1.\ k_1\ x_1\ \textsf{\textbf{in}}\ \square ::\ \square\ \kappa ::\ \mathcal{K}[\![\mathsf{K}]\!]) & = & \text{by Def } \textsc{Plug} \\
\textsc{Plug}((\lambda k.\ k\ \mathcal{V}[\![v]\!])\ (\lambda x_1.\ \lambda k_1.\ k_1\ x_1),\ \square\ \kappa ::\ \mathcal{K}[\![\mathsf{K}]\!]) & \rightarrow & \text{by } \beta\text{-reduction} \\
\textsc{Plug}((\lambda x_1.\ \lambda k_1.\ k_1\ x_1)\ \mathcal{V}[\![v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![\mathsf{K}]\!]) & \rightarrow & \text{by } \beta\text{-reduction} \\
\textsc{Plug}(\lambda k_1.\ k_1\ \mathcal{V}[\![v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![\mathsf{K}]\!]) & = & \text{by Def } \mathcal{S}[\![\cdot]\!] \\
\textsc{Plug}(\mathcal{S}[\![\textsf{\textbf{return}}\ v]\!],\ \square\ \kappa ::\ \mathcal{K}[\![\mathsf{K}]\!]) & = & \text{by Def } \mathcal{M}[\![\cdot]\!] \\
\mathcal{M}[\![\, \langle\, \textsf{\textbf{return}}\ v \parallel \mathsf{K} \,\rangle \,]\!] & &
\end{array}$$

Here we have used that k is fresh.

case *try*

In the following we have $\rho = \mathcal{R}\llbracket\ \mathsf{K}\ \rrbracket$, $u = l :: \rho$, $w = l :: \bullet$ and $v = \textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}$. We use $\mathcal{T}\llbracket\ u\ \rrbracket = \textsc{Cps}\ \mathcal{T}\llbracket\ \rho\ \rrbracket\ \mathcal{T}\llbracket\ \tau\ \rrbracket$ where τ is the type of the statement, $\mathcal{E}\llbracket\ w\ \rrbracket = \textsc{Lift}$ and $\mathcal{V}\llbracket\ v\ \rrbracket = \lambda x.\ \lambda k.\ \mathcal{S}\llbracket\ s\ \rrbracket$. Hence we have

$$\mathcal{M}\llbracket\ \langle\ \textbf{try}\ \{\ [r\ ;\ n](c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}\ \|\ \mathsf{K}\ \rangle\ \rrbracket \qquad = \quad \text{by Def } \mathcal{M}\llbracket\ \cdot\ \rrbracket$$

$$\textsc{Plug}(\mathcal{S}\llbracket\ \textbf{try}\ \{\ [r\ ;\ n](c) \Rightarrow s_0\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Def } \mathcal{S}\llbracket\ \cdot\ \rrbracket$$

$$\textsc{Plug}(\textsc{Reset}((\Lambda r.\ \lambda n.\ \lambda c.\ \mathcal{S}\llbracket\ s_0\ \rrbracket)\ (\textsc{Cps}\ \mathcal{T}\llbracket\ \rho\ \rrbracket\ \mathcal{T}\llbracket\ \tau\ \rrbracket)\ \textsc{Lift}\ (\lambda x.\ \lambda k.\ \mathcal{S}\llbracket\ s\ \rrbracket)),$$
$$\Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by above}$$

$$\textsc{Plug}(\textsc{Reset}((\Lambda r.\ \lambda n.\ \lambda c.\ \mathcal{S}\llbracket\ s_0\ \rrbracket)\ \mathcal{T}\llbracket\ u\ \rrbracket\ \mathcal{E}\llbracket\ w\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket),$$
$$\Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Def } \textsc{Reset}$$

$$\textsc{Plug}(((\Lambda r.\ \lambda n.\ \lambda c.\ \mathcal{S}\llbracket\ s_0\ \rrbracket)\ \mathcal{T}\llbracket\ u\ \rrbracket\ \mathcal{E}\llbracket\ w\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket)\ (\lambda x_1.\ \lambda k_1.\ k_1\ x_1),$$
$$\Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad \to^3 \quad \text{by } \beta\text{-reduction}$$

$$\textsc{Plug}((\mathcal{S}\llbracket\ s_0\ \rrbracket\ [r \mapsto \mathcal{T}\llbracket\ u\ \rrbracket,\ n \mapsto \mathcal{E}\llbracket\ w\ \rrbracket,\ c \mapsto \mathcal{V}\llbracket\ v\ \rrbracket])\ (\lambda x_1.\ \lambda k_1.\ k_1\ x_1),$$
$$\Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Def } \textsc{Plug}$$

$$\textsc{Plug}(\mathcal{S}\llbracket\ s_0\ \rrbracket\ [r \mapsto \mathcal{T}\llbracket\ u\ \rrbracket,\ n \mapsto \mathcal{E}\llbracket\ w\ \rrbracket,\ c \mapsto \mathcal{V}\llbracket\ v\ \rrbracket],$$
$$\Box\ \kappa ::\ \textbf{let}\ \kappa = \lambda x_1.\ \lambda k_1.\ k_1\ x_1\ \textbf{in}\ \Box ::\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Defs } \mathcal{K}\llbracket\ \cdot\ \rrbracket,\ \mathcal{F}\llbracket\ \cdot\ \rrbracket$$

$$\textsc{Plug}(\mathcal{S}\llbracket\ s_0\ \rrbracket\ [r \mapsto \mathcal{T}\llbracket\ u\ \rrbracket,\ n \mapsto \mathcal{E}\llbracket\ w\ \rrbracket,\ c \mapsto \mathcal{V}\llbracket\ v\ \rrbracket],$$
$$\Box\ \kappa ::\ \mathcal{K}\llbracket\ \#_l ::\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Lemma } \textsc{Subst}$$

$$\textsc{Plug}(\mathcal{S}\llbracket\ s_0[r \mapsto u,\ n \mapsto w,\ c \mapsto v]\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \#_l ::\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Def } \mathcal{M}\llbracket\ \cdot\ \rrbracket$$

$$\mathcal{M}\llbracket\ \langle\ s_0[r \mapsto u,\ n \mapsto w,\ c \mapsto v]\ \|\ \#_l ::\ \mathsf{K}\ \rangle\ \rrbracket$$

case *perform*

There are two cases to consider, one for $\textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}$ and one for $\textbf{cap}_l\ (\mathsf{S} ::\ \mathsf{H})$. Both cases are very similar, so for the common steps we simply write $\textbf{cap}_l\ h$ and correspondingly t in the translated term. Then we have

$$\mathcal{M}\llbracket\ \langle\ \textbf{do}\ (\textbf{cap}_l\ h)[e](v)\ \|\ \mathsf{K}\ \rangle\ \rrbracket \qquad = \quad \text{by Def } \mathcal{M}\llbracket\ \cdot\ \rrbracket$$

$$\textsc{Plug}(\mathcal{S}\llbracket\ \textbf{do}\ (\textbf{cap}_l\ h)[e](v)\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Def } \mathcal{S}\llbracket\ \cdot\ \rrbracket$$

$$\textsc{Plug}(\mathcal{E}\llbracket\ e\ \rrbracket\ \mathcal{T}\llbracket\ \tau_2\ \rrbracket\ t,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad = \quad \text{by Def } \mathcal{E}\llbracket\ \cdot\ \rrbracket$$

$$\textsc{Plug}((\Lambda a.\ \mathcal{E}'\llbracket\ e\ \rrbracket_a)\ \mathcal{T}\llbracket\ \tau_2\ \rrbracket\ t,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) \qquad \to \quad \text{by } \beta\text{-reduction}$$

$$\textsc{Plug}(\mathcal{E}'\llbracket\ e\ \rrbracket_{\mathcal{T}\llbracket\ \tau_2\ \rrbracket}\ t,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket)$$

Now for the case where $h = \{\ (x,\ k) \Rightarrow s\ \}$ we have

$$t = \mathcal{V}\llbracket\ \textbf{cap}_l\ h\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket = (\lambda x.\ \lambda k.\ \mathcal{S}\llbracket\ s\ \rrbracket)\ \mathcal{V}\llbracket\ v\ \rrbracket$$

and obtain

$$\textsc{Plug}(\mathcal{E}'[\![\ e\ ]\!]_{\mathcal{T}[\![\ \tau_2\ ]\!]}\ ((\lambda x.\ \lambda k.\ \mathcal{S}[\![\ s\ ]\!])\ \mathcal{V}[\![\ v\ ]\!]),\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) \quad \to \quad \text{by } \beta\text{-reduction}$$

$$\textsc{Plug}(\mathcal{E}'[\![\ e\ ]\!]_{\mathcal{T}[\![\ \tau_2\ ]\!]}\ ((\lambda k.\ \mathcal{S}[\![\ s\ ]\!])\ [x \mapsto \mathcal{V}[\![\ v\ ]\!]]),\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!])$$

Here we have used that $\mathcal{E}'[\![\ e\ ]\!]_a$ always is a value. For the case where $h\ =\ \mathsf{S} ::\ \mathsf{H}$ we have

$$t\ =\ \lambda k.\ \mathcal{V}[\![\ \mathbf{cap}_l\ h\ ]\!]\ \mathcal{V}[\![\ v\ ]\!]\ k\ =\ \lambda k.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!]\ k$$

Hence, in either case we now have an application of $\mathcal{E}'[\![\ e\ ]\!]_{\mathcal{T}[\![\ \tau_2\ ]\!]}$ to a value for which we write v_0. We use that e cannot contain variables so that we can use Lemma Perform and obtain

$$\begin{aligned}
&\textsc{Plug}(\mathcal{E}'[\![\ e\ ]\!]_{\mathcal{T}[\![\ \tau_2\ ]\!]}\ v_0,\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && \to^* && \text{by Lemma } \textsc{Perform}\\
&\textsc{Plug}(\mathcal{W}[\![\ N[\![\ e\ ]\!]\ ]\!]_{v_0},\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && = && \text{by Def } \mathcal{H}[\![\ \cdot\ ]\!]\\
&\textsc{Plug}(\mathcal{W}[\![\ N[\![\ e\ ]\!]\ ]\!]_{v_0},\ \Box\ \mathcal{H}[\![\ \bullet\ ]\!] ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && = && \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!]\\
&\mathcal{M}[\![\ \langle\ \mathbf{do}\ (\mathbf{cap}_l\ h)[\ N[\![\ e\ ]\!]\ ](v)\ \|\ \mathsf{K}\ \|\ \bullet\ \rangle\ ]\!]
\end{aligned}$$

case *unwind*

There are two cases to consider one for $\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}$ and one for $\mathbf{cap}_l\ \mathsf{H}$. But both cases proceed in exactly the same way and we again simply write $\mathbf{cap}_l\ h$ and correspondingly t in the translated term.

$$\begin{aligned}
&\mathcal{M}[\![\ \langle\ \mathbf{do}\ (\mathbf{cap}_l\ h)[l' ::\ w](v)\ \|\ \mathsf{S}_{l'} ::\ \mathsf{K}\ \|\ \mathsf{H}\ \rangle\ ]\!] && = && \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!]\\
&\textsc{Plug}(\mathcal{W}[\![\ l' ::\ w\ ]\!]_t,\ \Box\ \mathcal{H}[\![\ \mathsf{H}\ ]\!] ::\ \mathcal{K}[\![\ \mathsf{S}_{l'} ::\ \mathsf{K}\ ]\!]) && = && \text{by Def } \mathcal{W}[\![\ \cdot\ ]\!]\\
&\textsc{Plug}(\lambda k.\ \lambda j.\ \mathcal{W}[\![\ w\ ]\!]_t\ (\lambda x.\ k\ x\ j),\ \Box\ \mathcal{H}[\![\ \mathsf{H}\ ]\!] ::\ \mathcal{K}[\![\ \mathsf{S}_{l'} ::\ \mathsf{K}\ ]\!]) && = && \text{by Def } \mathcal{K}[\![\ \cdot\ ]\!]\\
&\textsc{Plug}(\lambda k.\ \lambda j.\ \mathcal{W}[\![\ w\ ]\!]_t\ (\lambda x.\ k\ x\ j),\ \Box\ \mathcal{H}[\![\ \mathsf{H}\ ]\!] ::\ \mathcal{F}[\![\ \mathsf{S}_{l'}\ ]\!] ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && = && \text{by Lemma } \textsc{PlugSeg}\\
&\textsc{Plug}(\lambda k.\ \lambda j.\ \mathcal{W}[\![\ w\ ]\!]_t\ (\lambda x.\ k\ x\ j),\\
&\qquad \Box\ \mathcal{H}[\![\ \mathsf{H}\ ]\!] ::\ \mathbf{let}\ \kappa\ =\ \mathcal{G}[\![\ \mathsf{S}_{l'}\ ]\!]\ \mathbf{in}\ \Box ::\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && = && \text{by Def } \textsc{Plug}\\
&\textsc{Plug}(((\lambda k.\ \lambda j.\ \mathcal{W}[\![\ w\ ]\!]_t\ (\lambda x.\ k\ x\ j))\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}_{l'}\ ]\!]])\ \kappa,\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && \to^2 && \text{by } \beta\text{-reduction}\\
&\textsc{Plug}(\mathcal{W}[\![\ w\ ]\!]_t\ (\lambda x.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}_{l'}\ ]\!]]\ x\ \kappa),\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && = && \text{by Def } \mathcal{H}[\![\ \cdot\ ]\!]\\
&\textsc{Plug}(\mathcal{W}[\![\ w\ ]\!]_t,\ \mathcal{H}[\![\ \mathsf{S}_{l'} ::\ \mathsf{H}\ ]\!],\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && = && \text{by Def } \textsc{Plug}\\
&\textsc{Plug}(\mathcal{W}[\![\ w\ ]\!]_t,\ \Box\ \mathcal{H}[\![\ \mathsf{S}_{l'} ::\ \mathsf{H}\ ]\!] ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) && = && \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!]\\
&\mathcal{M}[\![\ \langle\ \mathbf{do}\ (\mathbf{cap}_l\ h)[w](v)\ \|\ \mathsf{K}\ \|\ \mathsf{S}_{l'} ::\ \mathsf{H}\ \rangle\ ]\!]
\end{aligned}$$

Here we have used that $k,\ j$ are fresh, that κ occurs free in $\mathcal{H}[\![\ \mathsf{H}\ ]\!]$ and that $\mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}_{l'}\ ]\!]]$ always is a value by Lemma ResumpSubst.

case *handle*

$$\begin{aligned}
&\mathcal{M}\llbracket \langle \,\textbf{do cap}_l \; \{ \; (x,\; k) \Rightarrow s \;\}[\bullet](v) \parallel \mathsf{S}_l :: \; \mathsf{S}_{l'} :: \; \mathsf{K} \parallel \mathsf{H} \,\rangle \rrbracket && = \quad \text{by Def } \mathcal{M}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}(\mathcal{W}\llbracket \;\bullet\; \rrbracket_{((\lambda k.\; \mathcal{S}\llbracket\; s \;\rrbracket)[x \mapsto \mathcal{V}\llbracket\; v \;\rrbracket])},\; \square\; \mathcal{H}\llbracket \mathsf{H} \rrbracket :: \; \mathcal{K}\llbracket \mathsf{S}_l :: \; \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && = \quad \text{by Def } \mathcal{W}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}((\lambda k.\; \mathcal{S}\llbracket\; s \;\rrbracket)[x \mapsto \mathcal{V}\llbracket\; v \;\rrbracket],\; \square\; \mathcal{H}\llbracket \mathsf{H} \rrbracket :: \; \mathcal{K}\llbracket \mathsf{S}_l :: \; \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && = \quad \text{by Def } \mathcal{K}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}((\lambda k.\; \mathcal{S}\llbracket\; s \;\rrbracket)[x \mapsto \mathcal{V}\llbracket\; v \;\rrbracket],\; \square\; \mathcal{H}\llbracket \mathsf{H} \rrbracket :: \; \mathcal{F}\llbracket \mathsf{S}_l \rrbracket :: \; \mathcal{K}\llbracket \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && = \quad \text{by Lemma } \textsc{PlugSeg} \\
&\textsc{Plug}((\lambda k.\; \mathcal{S}\llbracket\; s \;\rrbracket)[x \mapsto \mathcal{V}\llbracket\; v \;\rrbracket], \\
&\qquad \square\; \mathcal{H}\llbracket \mathsf{H} \rrbracket :: \; \textbf{let}\ \kappa \;=\; \mathcal{G}\llbracket \mathsf{S}_l \rrbracket\ \textbf{in}\ \square :: \; \square\; \kappa :: \; \mathcal{K}\llbracket \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && = \quad \text{by Def } \textsc{Plug} \\
&\textsc{Plug}(((\lambda k.\; \mathcal{S}\llbracket\; s \;\rrbracket)[x \mapsto \mathcal{V}\llbracket\; v \;\rrbracket])\; \mathcal{H}\llbracket \mathsf{H} \rrbracket\; [\kappa \mapsto \mathcal{G}\llbracket \mathsf{S}_l \rrbracket], \\
&\qquad \square\; \kappa :: \; \mathcal{K}\llbracket \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && \to \quad \text{by Def } \mathcal{V}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}(((\lambda k.\; \mathcal{S}\llbracket\; s \;\rrbracket)[x \mapsto \mathcal{V}\llbracket\; v \;\rrbracket])\; \mathcal{V}\llbracket\, \textbf{cap}_{l'}\ (\mathsf{S}_l :: \; \mathsf{H}) \rrbracket, \\
&\qquad \square\; \kappa :: \; \mathcal{K}\llbracket \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && \to \quad \text{by } \beta\text{-reduction} \\
&\textsc{Plug}(\mathcal{S}\llbracket\; s \;\rrbracket\; [x \mapsto \mathcal{V}\llbracket\; v \;\rrbracket,\; k \mapsto \mathcal{V}\llbracket\, \textbf{cap}_{l'}\ (\mathsf{S}_l :: \; \mathsf{H}) \rrbracket], \\
&\qquad \square\; \kappa :: \; \mathcal{K}\llbracket \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && = \quad \text{by Lemma } \textsc{Subst} \\
&\textsc{Plug}(\mathcal{S}\llbracket\; s[x \mapsto v,\; k \mapsto \textbf{cap}_{l'}\ (\mathsf{S}_l :: \; \mathsf{H})] \;\rrbracket,\; \square\; \kappa :: \; \mathcal{K}\llbracket \mathsf{S}_{l'} :: \; \mathsf{K} \rrbracket) && = \quad \text{by Def } \mathcal{M}\llbracket \;\cdot\; \rrbracket \\
&\mathcal{M}\llbracket \langle \; s[x \mapsto v,\; k \mapsto \textbf{cap}_{l'}\ (\mathsf{S}_l :: \; \mathsf{H})] \parallel \mathsf{S}_{l'} :: \; \mathsf{K} \,\rangle \rrbracket
\end{aligned}$$

Here we have used that k only occurs free in s, that κ occurs free in $\mathcal{H}\llbracket \mathsf{H} \rrbracket$ and that $\mathcal{H}\llbracket \mathsf{H} \rrbracket\; [\kappa \mapsto \mathcal{G}\llbracket \mathsf{S}_l \rrbracket]$ always is a value by Lemma ResumpSubst.

case *rewindK*

$$\begin{aligned}
&\mathcal{M}\llbracket \langle \,\textbf{do}\ (\textbf{cap}_l\ (\mathsf{S}' :: \; \mathsf{H}'))[\bullet](v) \parallel \mathsf{S}_l :: \; \mathsf{K} \parallel \mathsf{S} :: \; \mathsf{H} \,\rangle \rrbracket && = \quad \text{by Def } \mathcal{M}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}(\mathcal{W}\llbracket \;\bullet\; \rrbracket_{(\lambda k.\; \mathcal{H}\llbracket\; \mathsf{H}' \;\rrbracket\; [\kappa \mapsto \mathcal{G}\llbracket\; \mathsf{S}' \;\rrbracket]\; \mathcal{V}\llbracket\; v \;\rrbracket\; k)}, \\
&\qquad \square\; \mathcal{H}\llbracket \mathsf{S} :: \; \mathsf{H} \rrbracket :: \; \mathcal{K}\llbracket \mathsf{S}_l :: \; \mathsf{K} \rrbracket) && = \quad \text{by Def } \mathcal{K}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}(\mathcal{W}\llbracket \;\bullet\; \rrbracket_{(\lambda k.\; \mathcal{H}\llbracket\; \mathsf{H}' \;\rrbracket\; [\kappa \mapsto \mathcal{G}\llbracket\; \mathsf{S}' \;\rrbracket]\; \mathcal{V}\llbracket\; v \;\rrbracket\; k)}, \\
&\qquad \square\; \kappa :: \; \textbf{let}\ \kappa \;=\; \mathcal{H}\llbracket \mathsf{S} :: \; \mathsf{H} \rrbracket\ \textbf{in}\ \square :: \; \mathcal{F}\llbracket \mathsf{S}_l \rrbracket :: \; \mathcal{K}\llbracket \mathsf{K} \rrbracket) && = \quad \text{by Defs } \mathcal{K}\llbracket \;\cdot\; \rrbracket,\ \mathcal{F}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}(\mathcal{W}\llbracket \;\bullet\; \rrbracket_{(\lambda k.\; \mathcal{H}\llbracket\; \mathsf{H}' \;\rrbracket\; [\kappa \mapsto \mathcal{G}\llbracket\; \mathsf{S}' \;\rrbracket]\; \mathcal{V}\llbracket\; v \;\rrbracket\; k)}, \\
&\qquad \square\; \kappa :: \; \mathcal{K}\llbracket\, (\textbf{resume}(\mathsf{S} :: \; \mathsf{H})\; \square :: \; \mathsf{S}_l) :: \; \mathsf{K} \rrbracket) && = \quad \text{by Def } \mathcal{H}\llbracket \;\cdot\; \rrbracket \\
&\textsc{Plug}(\mathcal{W}\llbracket \;\bullet\; \rrbracket_{(\lambda k.\; \mathcal{H}\llbracket\; \mathsf{H}' \;\rrbracket\; [\kappa \mapsto \mathcal{G}\llbracket\; \mathsf{S}' \;\rrbracket]\; \mathcal{V}\llbracket\; v \;\rrbracket\; k)}, \\
&\qquad \square\; \mathcal{H}\llbracket \;\bullet\; \rrbracket :: \; \mathcal{K}\llbracket\, (\textbf{resume}(\mathsf{S} :: \; \mathsf{H})\; \square :: \; \mathsf{S}_l) :: \; \mathsf{K} \rrbracket) && = \quad \text{by Def } \mathcal{M}\llbracket \;\cdot\; \rrbracket \\
&\mathcal{M}\llbracket \langle \,\textbf{do}\ (\textbf{cap}_l\ (\mathsf{S}' :: \; \mathsf{H}'))[\bullet](v) \parallel \\
&\qquad (\textbf{resume}(\mathsf{S} :: \; \mathsf{H})\; \square :: \; \mathsf{S}_l) :: \; \mathsf{K} \parallel \;\bullet\; \rangle \rrbracket
\end{aligned}$$

case *handleK*

$$\begin{array}{ll}
\mathcal{M}[\![\ \langle\ \textbf{do}\ (\textbf{cap}_l\ (\mathsf{S} ::\ \mathsf{H}))[\bullet](v)\ \|\ \mathsf{S}_l ::\ \mathsf{K}\ \|\ \bullet\ \rangle\ ]\!] & = \quad \text{by Def } \mathcal{M}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{W}[\![\ \bullet\ ]\!]_{(\lambda k.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!]\ k)},\ \Box\ \mathcal{H}[\![\ \bullet\ ]\!] ::\ \mathcal{K}[\![\ \mathsf{S}_l ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \mathcal{H}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{W}[\![\ \bullet\ ]\!]_{(\lambda k.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!]\ k)},\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{S}_l ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \mathcal{W}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\lambda k.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!]\ k,\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{S}_l ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \textsc{Plug} \\
\textsc{Plug}((\lambda k.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!]\ k)\ \kappa,\ \mathcal{K}[\![\ \mathsf{S}_l ::\ \mathsf{K}\ ]\!]) & \to \quad \text{by } \beta\text{-reduction} \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!]\ \kappa,\ \mathcal{K}[\![\ \mathsf{S}_l ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \textsc{Plug} \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!],\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{S}_l ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Lemma } \textsc{Resume} \\
\mathcal{M}[\![\ \langle\ \textbf{resume}(\mathsf{S} ::\ \mathsf{H})(v)\ \|\ \mathsf{S}_l ::\ \mathsf{K}\ \rangle\ ]\!] &
\end{array}$$

Here we have used that k is fresh.

case *rewind*

$$\begin{array}{ll}
\mathcal{M}[\![\ \langle\ \textbf{resume}(\mathsf{S} ::\ \mathsf{S}' ::\ \mathsf{H})(v)\ \|\ \mathsf{K}\ \rangle\ ]\!] & = \quad \text{by Lemma } \textsc{Resume} \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{S}' ::\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!],\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \textsc{Plug} \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{S}' ::\ \mathsf{H}\ ]\!]\ \mathcal{V}[\![\ v\ ]\!], & \\
\qquad \textbf{let}\ \kappa\ =\ \mathcal{G}[\![\ \mathsf{S}\ ]\!]\ \textbf{in}\ \Box ::\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = \quad \text{by Lemma } \textsc{PlugSeg} \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{S}' ::\ \mathsf{H}\ ]\!]\ \mathcal{V}[\![\ v\ ]\!],\ \mathcal{F}[\![\ \mathsf{S}\ ]\!] ::\ \mathcal{K}[\![\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \mathcal{K}[\![\ \cdot\ ]\!] \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{S}' ::\ \mathsf{H}\ ]\!]\ \mathcal{V}[\![\ v\ ]\!],\ \mathcal{K}[\![\ \mathsf{S} ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \mathcal{H}[\![\ \cdot\ ]\!] \\
\textsc{Plug}((\lambda x_0.\ \mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}'\ ]\!]]\ x_0\ \kappa)\ \mathcal{V}[\![\ v\ ]\!],\ \mathcal{K}[\![\ \mathsf{S} ::\ \mathsf{K}\ ]\!]) & \to \quad \text{by } \beta\text{-reduction} \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}'\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!]\ \kappa,\ \mathcal{K}[\![\ \mathsf{S} ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Def } \textsc{Plug} \\
\textsc{Plug}(\mathcal{H}[\![\ \mathsf{H}\ ]\!]\ [\kappa \mapsto \mathcal{G}[\![\ \mathsf{S}'\ ]\!]]\ \mathcal{V}[\![\ v\ ]\!],\ \Box\ \kappa ::\ \mathcal{K}[\![\ \mathsf{S} ::\ \mathsf{K}\ ]\!]) & = \quad \text{by Lemma } \textsc{Resume} \\
\mathcal{M}[\![\ \langle\ \textbf{resume}(\mathsf{S}' ::\ \mathsf{H})(v)\ \|\ \mathsf{S} ::\ \mathsf{K}\ \rangle\ ]\!] &
\end{array}$$

Here have we used that x_0 is fresh and that κ occurs free in $\mathcal{H}[\![\ \mathsf{S}' ::\ \mathsf{H}\ ]\!]$.

case *resume-1*

$$\begin{array}{lll}
\mathcal{M}\llbracket\, \langle\, \textbf{resume}((\textbf{val}\ x\ =\ \Box;\ s :: \ \mathsf{S}) ::\ \bullet)(v) \parallel \mathsf{K}\,\rangle\, \rrbracket & = & \text{by Lemma } \textsc{Resume} \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \bullet\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \textbf{val}\ x\ =\ \Box;\ s :: \ \mathsf{S}\ \rrbracket]\ \mathcal{V}\llbracket\ v\ \rrbracket, & & \\
\qquad \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{H}\llbracket\ \cdot\ \rrbracket \\[1em]
\textsc{Plug}(\mathcal{G}\llbracket\ \textbf{val}\ x\ =\ \Box;\ s :: \ \mathsf{S}\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{G}\llbracket\ \cdot\ \rrbracket \\[1em]
\textsc{Plug}((\lambda x.\ \mathcal{S}\llbracket\ s\ \rrbracket\ \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket)\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \textsc{Plug} \\[1em]
\textsc{Plug}((\lambda x.\ \mathcal{S}\llbracket\ s\ \rrbracket\ \kappa)\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \textbf{let}\ \kappa\ =\ \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket\ \textbf{in}\ \Box ::\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Lemma } \textsc{PlugSeg} \\[1em]
\textsc{Plug}((\lambda x.\ \mathcal{S}\llbracket\ s\ \rrbracket\ \kappa)\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \mathcal{F}\llbracket\ \mathsf{S}\ \rrbracket ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{K}\llbracket\ \cdot\ \rrbracket \\[1em]
\textsc{Plug}((\lambda x.\ \mathcal{S}\llbracket\ s\ \rrbracket\ \kappa)\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & \to & \text{by } \beta\text{-reduction} \\[1em]
\textsc{Plug}(\mathcal{S}\llbracket\ s\ \rrbracket\ [x \mapsto \mathcal{V}\llbracket\ v\ \rrbracket]\ \kappa,\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \textsc{Plug} \\[1em]
\textsc{Plug}(\mathcal{S}\llbracket\ s\ \rrbracket\ [x \mapsto \mathcal{V}\llbracket\ v\ \rrbracket],\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & = & \text{by Lemma } \textsc{Subst} \\[1em]
\textsc{Plug}(\mathcal{S}\llbracket\ s\ [x \mapsto v]\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{M}\llbracket\ \cdot\ \rrbracket \\[1em]
\mathcal{M}\llbracket\ \langle\ s\ [x \mapsto v] \parallel \mathsf{S} ::\ \mathsf{K}\ \rangle\ \rrbracket & &
\end{array}$$

Here we have used that x occurs free only in s.

case *resume-2*

$$\begin{array}{lll}
\mathcal{M}\llbracket\ \langle\ \textbf{resume}((\textbf{resume}(\mathsf{S}' ::\ \mathsf{H})\ \Box ::\ \mathsf{S}) ::\ \bullet)(v) \parallel \mathsf{K}\ \rangle\ \rrbracket & = & \text{by Lemma } \textsc{Resume} \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \bullet\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \textbf{resume}(\mathsf{S}' ::\ \mathsf{H})\ \Box ::\ \mathsf{S}\ \rrbracket]\ \mathcal{V}\llbracket\ v\ \rrbracket, & & \\
\qquad \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{H}\llbracket\ \cdot\ \rrbracket \\[1em]
\textsc{Plug}(\mathcal{G}\llbracket\ \textbf{resume}(\mathsf{S}' ::\ \mathsf{H})\ \Box ::\ \mathsf{S}\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{G}\llbracket\ \cdot\ \rrbracket \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \mathsf{S}' ::\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket]\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \textsc{Plug} \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \mathsf{S}' ::\ \mathsf{H}\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \textbf{let}\ \kappa\ =\ \mathcal{G}\llbracket\ \mathsf{S}\ \rrbracket\ \textbf{in}\ \Box ::\ \Box\ \kappa,\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Lemma } \textsc{PlugSeg} \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \mathsf{S}' ::\ \mathsf{H}\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \mathcal{F}\llbracket\ \mathsf{S}\ \rrbracket ::\ \mathcal{K}\llbracket\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{K}\llbracket\ \cdot\ \rrbracket \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \mathsf{S}' ::\ \mathsf{H}\ \rrbracket\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \mathcal{H}\llbracket\ \cdot\ \rrbracket \\[1em]
\textsc{Plug}((\lambda x_0.\ \mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}'\ \rrbracket]\ x_0\ \kappa)\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & \to & \text{by } \beta\text{-reduction} \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}'\ \rrbracket]\ \mathcal{V}\llbracket\ v\ \rrbracket\ \kappa,\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & = & \text{by Def } \textsc{Plug} \\[1em]
\textsc{Plug}(\mathcal{H}\llbracket\ \mathsf{H}\ \rrbracket\ [\kappa \mapsto \mathcal{G}\llbracket\ \mathsf{S}'\ \rrbracket]\ \mathcal{V}\llbracket\ v\ \rrbracket,\ \Box\ \kappa ::\ \mathcal{K}\llbracket\ \mathsf{S} ::\ \mathsf{K}\ \rrbracket) & = & \text{by Lemma } \textsc{Resume} \\[1em]
\mathcal{M}\llbracket\ \langle\ \textbf{resume}(\mathsf{S}' ::\ \mathsf{H})(v) \parallel \mathsf{S} ::\ \mathsf{K}\ \rangle\ \rrbracket & &
\end{array}$$

Here we have used that x_0 is fresh that κ occurs free in $\mathcal{H}\llbracket\ \mathsf{S}' ::\ \mathsf{H}\ \rrbracket$.

case *resume-3*

$\mathcal{M}[\![\langle \textsf{\textbf{resume}}(\#_l :: \bullet)(v) \parallel \mathsf{K} \rangle]\!]$	$=$	by Lemma Resume
$\textsc{Plug}(\mathcal{H}[\![\bullet]\!] [\kappa \mapsto \mathcal{G}[\![\#_l]\!]] \mathcal{V}[\![v]\!], \Box\, \kappa :: \mathcal{K}[\![\mathsf{K}]\!])$	$=$	by Def $\mathcal{H}[\![\cdot]\!]$
$\textsc{Plug}(\mathcal{G}[\![\#_l]\!] \mathcal{V}[\![v]\!], \Box\, \kappa :: \mathcal{K}[\![\mathsf{K}]\!])$	$=$	by Def $\mathcal{G}[\![\cdot]\!]$
$\textsc{Plug}((\lambda x.\ \lambda k.\ k\ x)\ \mathcal{V}[\![v]\!], \Box\, \kappa :: \mathcal{K}[\![\mathsf{K}]\!])$	$\to$	by β-reduction
$\textsc{Plug}(\lambda k.\ k\ \mathcal{V}[\![v]\!], \Box\, \kappa :: \mathcal{K}[\![\mathsf{K}]\!])$	$=$	by Def $\mathcal{M}[\![\cdot]\!]$
$\mathcal{M}[\![\langle \textsf{\textbf{return}}\ v \parallel \mathsf{K} \rangle]\!]$		

◀

B.3.2 Typability Preservation

Finally, we briefly consider which changes are necessary in the proof of typability preservation for the CPS translation (Theorem 3).

▶ **Theorem 18** (Typability Preservation for the CPS Translation).
If $\Gamma \mid \rho \vdash s : \tau$, *then* $\mathcal{T}[\![\Gamma]\!] \vdash \mathcal{S}[\![s]\!] : \textsc{Cps}\ \mathcal{T}[\![\rho]\!]\ \mathcal{T}[\![\tau]\!]$.
If $\Gamma \vdash v : \tau$, *then* $\mathcal{T}[\![\Gamma]\!] \vdash \mathcal{V}[\![v]\!] : \mathcal{T}[\![\tau]\!]$.

Proof. The typability preservation for the CPS translation from Λ_{cap} to System F is almost the same as before. The only changes are that continuations are typed as **Cap** $\rho\ \tau_1\ \tau_2$ instead of $\tau_1 \to_\rho \tau_2$ and the translation performs an η-extension. But an η-extension does not change the type and the two above types are translated to the same type in System F. Hence, the original proof carries over almost unchanged. ◀