Zero-cost Effect Handlers by Staging (Technical Report)

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

Effect handlers offer ways to structure programs with complex control-flow patterns. Yet, current implementations of effect handlers incur a significant price in performance for doing so. We present a language that makes it possible to eliminate the overhead introduced by using effect handlers to abstract over effect operations. We present a translation of this language to simply typed lambda calculus in continuation passing style that preserves the well-typedness of terms. We present a second translation that, guided by automatically inserted staging annotations, guarantees to eliminate abstractions and applications related to effect handlers. We implement the translations and generate code in multiple languages. We validate the efficiency of the generated code theoretically, by proving that the use of certain language constructs never leads to run-time reductions, and practically, with a suite of benchmarks comparing with existing implementations of effect handlers and control operators.

1 INTRODUCTION

Effect handlers [43, 44] offer high-level control-flow abstractions that are user definable and composable. They have been successfully used to develop libraries for asynchronous programming, libraries for concurrent system programming, programming with coroutines, stream processing, and many more [8, 15, 17, 21, 24, 32, 42]. Effect handlers naturally allow users to combine these libraries and the corresponding domain-specific abstractions in one program.

As of today, this extra abstraction comes with a cost in performance [31, 46]. There are two different aspects to the performance of such abstractions: enabling compile time optimization [46, 52] and optimizing the language runtime [31]. In this work, we are only concerned with the former, with the ultimate goal to fully eliminate the abstraction overhead of effect handlers at compile time. Efficient implementations of effect handlers would enable programmers to develop many general-purpose or domain-specific control flow constructs as libraries without sacrificing performance.

The meaning of effectful programs depends on their evaluation context [51]. In languages with support for effect handlers, the handler implementations are part of this evaluation context. Typically, language runtimes performs a dynamic lookup to find a matching handler implementation for an effect operation. These dynamic lookups incur a run-time penalty and, more importantly to this work, they preclude compile-time optimizations. To evaluate the call to an effect operation typically includes two tasks at runtime: firstly, performing a linear lookup through the evaluation context to find the corresponding effect handler. Secondly, capturing a segment of the context delimited by that very handler [16, 22, 24, 33]. In general, the full evaluation context can only be known at run time. However, if certain information about the context is available at compile time, we can use it to specialize effectful programs.

We present λ_{Cap}, a calculus that operates under the assumptions that the following information is statically known for a given effectful computation: (a) all used handler implementations and (b) the order in which they enclose the effectful computation. This assumption limits the kinds of programs that can be expressed in λ_{Cap}. For example, it prevents us from abstracting over handlers and from handling a program with an unbounded number of handlers. These restrictions help

to understand the core assumptions necessary to guarantee efficient code generation while still allowing many programs using effect handlers to be expressed.

λ_{Cap} makes enough information explicit to guarantee full elimination of the abstraction overhead introduced by handlers. It incorporates the two kinds of static information in the following way. Firstly, it supports effect handlers in *explicit capability-passing style* [5, 6, 53]. By making the flow of capabilities explicit and marking them as static, we know at compile-time which effect operation corresponds to which occurrence of a handler. Secondly, the type-system of λ_{Cap} tracks the *stack shape* of an effectful computation: a list of types corresponding to the types expected by enclosing handlers. Guided by the stack shape, we then perform an iterated CPS translation [12, 22, 49]. Statically knowing the handler for each effect operation and statically knowing the stack shape allow us to reduce all abstractions related to handlers at compile time. We preserve all function abstractions and applications that were present in the original program.

To assess the performance of our compilation strategy, we compare code generated from λ_{Cap} with Koka [33], Multicore OCaml [16], and Chez Scheme [18]. The benchmarks indicate that our translation offers significant speedups (of up to 256x) for examples, which heavily use effects and handlers and shows competitive performance for examples with only simple uses of effect handlers. Implementing these examples also shows that, despite the aforementioned restrictions, λ_{Cap} can express common usages of effect handlers.

Specifically, we make the following contributions:

- We present the formal language λ_{Cap} with effect handlers in explicit capability-passing style. The language treats capabilities as second class to guarantee inlining in the translation (Section 3.1).
- We present a translation from λ_{Cap} to STLC (Section 4) that preserves well-typedness (Theorems 5.1 and 5.3). This entails effect safety (Theorem 5.2): pure programs do not have unhandled effects at run time.
- We guarantee that translated programs are free of overhead introduced by the handler abstraction (Theorem 5.5).
- We have implemented λ_{Cap} and performed benchmarks, which suggest that the code we generate is competitive with or faster than Koka, Multicore OCaml, and Chez Scheme code using effects (Section 5).

2 OVERVIEW

In this section, we present a brief informal overview of our approach, which will then be explained in detail in the subsequent sections.

When programming with effect handlers, we write effectful functions using effect operations. Figure 1a shows an example program, adapted from Danvy and Filinski [12], and written in λ_{Cap} [1]. The effectful function choice uses the two capabilities flip and fail to choose a number between the given argument n and 1. If the parameter n is smaller than 1, choice fails. Otherwise it flips a coin to decide if it immediately returns n or recursively calls itself with a decremented argument. The signature of effect operations is given by the following two global signatures.

```
effect Flip : () → Bool        effect Fail : () → Void
```

We write the function choice in capability-passing style. That is, it explicitly abstracts over the *capabilities* flip and fail. Capabilities are second class [40] – they cannot be closed over or returned

[1] We omit type annotations and define top-level functions, not available in the core calculus.

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

def handledChoice(n) {
  handle flip = Flip((), k) ⇒
      append(do k(True), do k(False)) in
    handle fail = Fail((), k) ⇒ Nil in
      Cons(choice[lift flip, fail](n), Nil)
}
```

(a) Example source program in capability-passing style.

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

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

(b) Generated code in iterated CPS and inlined handlers.

Fig. 1. Running example in our language λ_{Cap} and its translation into CPS.

from functions. In the body of the function, we use the capabilities (e.g., do flip()) to call an effect operation.

Handling effects. To give meaning to effect operations, we enclose effectful programs in handlers. For example, we can implement Flip and Fail to gather all choices into a list. The function handledChoice does exactly this. Handlers are written as handle ... in ... and provide capabilities. In our example, the handlers for Flip and Fail in function handledChoice bind capabilities to the names flip and fail, which we explicitly pass to the call of choice. To implement an effect operation, a handler gets access to the *current continuation* at the invocation of the effect operation. At the same time, the handler acts as a *delimiter* for these continuations. In our example, the handler for Flip calls the continuation k twice, once with True and once with False. It expects the results of these two calls to be lists, appends them, and answers with the appended list. The implementation for fail ignores k and immediately answers with the empty list.

Effect safety. The *answer type* is the return type of the computation that a handler encloses [12]. The *stack shape* of a computation describes the list of answer types at its enclosing handlers, from outermost to innermost. In our example, both handlers have the same answer type IntList and the stack shape at the invocation of choice is thus IntList, IntList. To achieve answer-type safety (i.e., capturing and applying the continuation is type safe) and effect safety (i.e., all effects are eventually handled), the type system of λ_{Cap} indexes the types of capabilities and the types of effectful functions by the stack shape they assume. Adapting notation by Zhang and Myers [53], we write the type of the choice function as:

$$\overbrace{[\mathsf{Flip}]_{\mathsf{IntList},\mathsf{IntList}} \to [\mathsf{Fail}]_{\mathsf{IntList},\mathsf{IntList}}}^{\text{capabilities}} \to \mathsf{Int} \to \overbrace{[\mathsf{Int}]_{\mathsf{IntList},\mathsf{IntList}}}^{\text{effectful return type}}$$

It is an effectful function that takes a capability for Flip, a capability for Fail, and an Int, and returns an Int. It assumes a stack shape IntList, IntList. To safely invoke an effect operation, the stack shape of the computation at the invocation site and the stack shape of the capability have to agree. Since we created the capability flip at the outer handler, its type is $[\ \mathsf{Flip}\ ]_{\mathsf{IntList}}$. To use it inside of the

inner handler, as an argument to choice, our effect system requires us to explicitly adapt it using lift. This way, the capability can be used in a context with the larger stack shape IntList, IntList.

Compilation of λ_{Cap}. We translate our source language λ_{Cap} to STLC in iterated continuation-passing style (CPS) [12]. Directed by the statically known stack shape, our translation introduces one continuation argument for every delimiting handler. Specializing choice to the stack shape IntList, IntList, we obtain:

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

The generated code uses two continuations corresponding to the two delimiters for Flip and Fail. While it is specialized to the stack shape, the translation of choice still abstracts over capabilities flip and fail. To also specialize the code to the concrete handler implementations, we use staging annotations and reduce capability abstractions and applications statically. This way, the implementations of Flip and Fail provided by the corresponding handlers are inlined into the body of choice. Figure 1b shows the final code we generate for this example. We chose the name choiceFlipFail for this specialization to the handler implementations in this example. The cost of the handler abstraction has been fully removed and the function is specialized to both the effect operation implementations and the stack shape at its call-site. There is no explicit runtime search for a matching handler like in Koka [33] or Eff [44]. Instead, the implementations of the effect operations have been inlined into the body of choice. There is no search for a delimiter on the stack either. We directly invoke the corresponding continuation.

To sum up our approach: Programs are written in explicit capability-passing style. Effectful functions and capabilities are indexed by the stack shape. Capabilities need to be lifted explicitly to adjust them to the stack shape as required. Using this information and guaranteeing that capabilities can always be inlined, we specialize functions to their context and remove the cost associated with the handler abstraction.

3 THE LANGUAGE λ_{Cap}

In this section, we formally introduce λ_{Cap} – a language with effects, handlers, and capabilities. It follows the calculus by Zhang and Myers [53] with some notable differences discussed in Section 7.1.

3.1 Syntax of Terms

Figure 2 defines the syntax of λ_{Cap}. Like other presentations of languages with effect handlers [22, 25, 45], our language is based on a fine-grain call-by-value lambda calculus [36]. That is, we syntactically distinguish between *expressions* (often also referred to as "values") and *statements* (also called "computations"). Only statements can have effects. In this sense, our expressions are "trivial" while statements are "serious" [47]. Other than most effect languages, which do not represent capabilities explicitly, we also syntactically distinguish between expressions and capabilities. Capabilities are second class and cannot be returned from functions. As we will see, this separation is important to guarantee that handler implementations can be fully inlined at compile-time.

Syntax of Terms:

Expressions

e	::=	True | False | ...	primitive constants
	|	x	term variables
	|	$(x : \tau) \Rightarrow s$	lambda abstraction
	|	$\text{fix}\, f\, (x : \tau) \Rightarrow s$	recursive abstraction
	|	$[c : [\![\mathbb{F}]\!]_{\overline{\tau}}] \Rightarrow e$	capability abstraction
	|	$e[h]$	capability application

Statements

s	::=	$e(e)$	application
	|	$\text{val}\, x \leftarrow s;\ s$	sequence
	|	$\text{return}\, e$	return
	|	$\text{do}\, h(e)$	effect call
	|	$\text{handle}\, c = h\, \text{in}\, s$	effect handler

Capabilities

h	::=	$c \mid k$	capability variables
	|	$\mathbb{F}(x,\ k) \Rightarrow s$	handler implementation
	|	$\text{lift}\, h$	lifted capability

Syntax of Types:

Dynamic Types

τ	::=	Int | Bool | ...	base types
	|	$\tau \rightarrow [\![\tau]\!]_{\overline{\tau}}$	effectful function type

Static Types

σ	::=	$[\![\mathbb{F}]\!]_{\overline{\tau}} \rightarrow \sigma$	capability function type
	|	τ	dynamic type

Operation Names

$\mathbb{F}$ ::= Flip | Fail | Emit | Resume_i | ...

Operation Signatures

Σ ::= $\emptyset$ | $\Sigma,\ \mathbb{F} : \tau \rightarrow \tau'$

Type Environment

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

Capability Environment

Θ ::= $\emptyset$ | $\Theta,\ c : [\![\mathbb{F}]\!]_{\overline{\tau}}$

Stack Shape

$\overline{\tau}$ ::= $\emptyset$ | $\overline{\tau},\ \tau$

Fig. 2. Syntax of terms and types (λ_{Cap}).

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

Statements. Both, function application (i.e., $e(e')$) and calling capabilities (i.e., $\text{do}\, h(e)$) are considered effectful. The latter corresponds to an effect call in other languages with effect handlers. Expressions are embedded into statements with $\text{return}\, e$ and we use the syntax $\text{val}\, x \leftarrow s;\ s'$ to sequence the evaluation of the two statements s and s'. The result of s is available in s' under the name x. Finally, we handle effectful programs with $\text{handle}\, c = h\, \text{in}\, s$. The capability variable c will be bound to the handler implementation h in the statement s. It also installs a delimiter for the continuation that is captured when c is used.

Capabilities. To separate expression variables from capability variables, the latter are drawn from a different namespace (e.g., flip, fail, or k). Similarly, we use the meta-variables x for term variables and c and k for capability variables. The $\text{lift}\, h$ construct adjusts a capability h to be compatible to a larger stack shape. Capabilities are handler implementations constructed with $\mathbb{F}(x,\ k) \Rightarrow s$. The argument x and the continuation k are bound in the implementation of the effect operation given by s. As we will see, we model continuations as capabilities and thus k has to be invoked with $\text{do}\, k(e)$.

3.2 Type System of λ_{Cap}

Similarly to the syntactic separation of expressions and capabilities, in the syntax of types we distinguish between *dynamic types* and *static types* (Figure 2). Later, terms of static types will disappear during translation while terms of dynamic types will appear in the generated program.

Dynamic types include base types (e.g., Int and Bool) and effectful function types $\tau \to [\ \tau'\ ]_{\overline{\tau}}$. There are two equally valid intuitions about effectful function types. One can read a function type as "Given an argument of type τ, the function produces a result τ', potentially using control effects in $\overline{\tau}$". However, there is also a second reading "Given τ, the function can only be called in a context with stack shape $\overline{\tau}$ to produce a result of type τ'". We will mostly apply the latter intuition. Stack shapes are comma separated lists of dynamic types τ, representing the types at the delimiters (i.e. handlers) from outermost to innermost. They serve a similar purpose like effect rows of Koka [33] or Links [21]. Like effect rows in Koka and Links, our stack shapes guarantee that our control effects are handled and all continuations are correctly delimited. However, unlike effect rows, stack shapes are *ordered*. As an example, the stack shape Int, String describes a context with an outer handler at type Int and an inner handler at type String.

Static types are sequences of capability parameters ending in a dynamic type. This ensures that all capability arguments come before any other arguments. Like effectful functions, the type of each capability parameter $[\ \mathbb{F}\ ]_{\overline{\tau}}$ is restricted to a specific stack shape $\overline{\tau}$. We use the meta-variable $\mathbb{F}$ to denote a globally fixed set of operation names and assume a global signature environment Σ that maps operation names to their input and output types. We model continuations as capabilities and include a family Resume_i in the set of operation names. Each syntactic occurrence of handle ... in ... induces a distinct operation name Resume_i. The typing of the corresponding use of handle fully determines the signature of Resume_i in Σ.

Following the distinction between expressions and capabilities, we also assume two separate environments. A type environment Γ that assigns variables x to dynamic types τ and a capability environment Θ that associates capability variables c with operation names $\mathbb{F}$ and stack shapes $\overline{\tau}$.

3.2.1 Typing Rules. The typing rules in Figure 3 are defined by three mutually recursive typing judgements – one for each syntactic category. Importantly, while the judgement form $\Theta \mid \Gamma \vdash_{\text{exp}} e : \sigma$ may assign a static type σ to expressions, all premises of our statement typing rules require expressions to be typed against a dynamic type τ. This way, we make sure that all effectful functions are always fully applied to the corresponding capabilities. The judgement form $\Theta \mid \Gamma \vdash_{\text{stm}} s : [\ \tau\ ]_{\overline{\tau}}$ assigns a pair of a dynamic type τ and a stack shape $\overline{\tau}$ to the statement s. Note that $[\ \tau\ ]_{\overline{\tau}}$ is not one type but two separate outputs of the judgement.

The typing rules include standard rules for variables (Var), abstraction (Lam), recursive abstraction (Fix), and application (App). Sequencing with rule Val requires that the stack shapes of the two statements s and s' agree. Similarly in rule Do the stack shape of the used capability and the do statement have to agree.

We treat capabilities as second class [40]. That is, they can neither be closed over, nor can they be returned from a function. This becomes evident in the typing rules. In the rule Ret, the returned pure expression e has to be typed against a dynamic type τ. Expressions thus are always fully specialized (that is, applied to capabilities) before they can be returned. The resulting statement is compatible with any stack shape $\overline{\tau}$. Similarly, argument expressions in rule App are required to be of a dynamic type τ'. The rules for capability abstraction (Cap-Lam) and application (Cap-App) are similar to the corresponding rules for value abstraction and application. However, capability abstraction introduces the capability variable c in the second class environment Θ and capability application uses the capability typing judgement $\Theta \mid \Gamma \vdash_{\text{cap}} h : [\ \mathbb{F}\ ]_{\overline{\tau}}$ to check h against the operation name $\mathbb{F}$ in the stack shape $\overline{\tau}$.

Expression Typing. $\boxed{\Theta \mid \Gamma \vdash_{\text{exp}} e : \sigma}$

$$\frac{\Gamma(x) = \tau}{\Theta \mid \Gamma \vdash_{\text{exp}} x : \tau}\ [\textsc{Var}]$$

$$\frac{\Theta \mid \Gamma,\ x : \tau \vdash_{\text{stm}} s : [\,\tau'\,]_{\overline{\tau}}}{\Theta \mid \Gamma \vdash_{\text{exp}} (x : \tau) \Rightarrow s : \tau \to [\,\tau'\,]_{\overline{\tau}}}\ [\textsc{Lam}] \qquad \frac{\Theta \mid \Gamma,\ f : \tau \to [\,\tau'\,]_{\overline{\tau}},\ x : \tau \vdash_{\text{stm}} s : [\,\tau'\,]_{\overline{\tau}}}{\Theta \mid \Gamma \vdash_{\text{exp}} \text{fix}\, f\, (x : \tau) \Rightarrow s : \tau \to [\,\tau'\,]_{\overline{\tau}}}\ [\textsc{Fix}]$$

$$\frac{\Theta,\ c : [\,\mathbb{F}\,]_{\overline{\tau}} \mid \Gamma \vdash_{\text{exp}} e : \sigma}{\Theta \mid \Gamma \vdash_{\text{exp}} [c : [\,\mathbb{F}\,]_{\overline{\tau}}] \Rightarrow e : [\,\mathbb{F}\,]_{\overline{\tau}} \to \sigma}\ [\textsc{Cap-Lam}] \qquad \frac{\Theta \mid \Gamma \vdash_{\text{exp}} e : [\,\mathbb{F}\,]_{\overline{\tau}} \to \sigma \quad \Theta \mid \Gamma \vdash_{\text{cap}} h : [\,\mathbb{F}\,]_{\overline{\tau}}}{\Theta \mid \Gamma \vdash_{\text{exp}} e[h] : \sigma}\ [\textsc{Cap-App}]$$

Statement Typing. $\boxed{\Theta \mid \Gamma \vdash_{\text{stm}} s : [\,\tau\,]_{\overline{\tau}}}$

$$\frac{\Theta,\ c : [\,\mathbb{F}\,]_{\overline{\tau},\,\tau} \mid \Gamma \vdash_{\text{stm}} s : [\,\tau\,]_{\overline{\tau},\,\tau} \qquad \Theta \mid \Gamma \vdash_{\text{cap}} h : [\,\mathbb{F}\,]_{\overline{\tau},\,\tau}}{\Theta \mid \Gamma \vdash_{\text{stm}} \text{handle}\ c = h\ \text{in}\ s : [\,\tau\,]_{\overline{\tau}}}\ [\textsc{Handle}]$$

$$\frac{\Theta \mid \Gamma \vdash_{\text{stm}} s : [\,\tau\,]_{\overline{\tau}} \qquad \Theta \mid \Gamma,\ x : \tau \vdash_{\text{stm}} s' : [\,\tau'\,]_{\overline{\tau}}}{\Theta \mid \Gamma \vdash_{\text{stm}} \text{val}\ x \leftarrow s;\ s' : [\,\tau'\,]_{\overline{\tau}}}\ [\textsc{Val}] \qquad \frac{\Theta \mid \Gamma \vdash_{\text{exp}} e : \tau}{\Theta \mid \Gamma \vdash_{\text{stm}} \text{return}\ e : [\,\tau\,]_{\overline{\tau}}}\ [\textsc{Ret}]$$

$$\frac{\Theta \mid \Gamma \vdash_{\text{exp}} e : \tau' \to [\,\tau\,]_{\overline{\tau}} \qquad \Theta \mid \Gamma \vdash_{\text{exp}} e' : \tau'}{\Theta \mid \Gamma \vdash_{\text{stm}} e(e') : [\,\tau\,]_{\overline{\tau}}}\ [\textsc{App}] \qquad \frac{\Theta \mid \Gamma \vdash_{\text{cap}} h : [\,\mathbb{F}\,]_{\overline{\tau}} \quad \Sigma(\mathbb{F}) = \tau' \to \tau \quad \Theta \mid \Gamma \vdash_{\text{exp}} e : \tau'}{\Theta \mid \Gamma \vdash_{\text{stm}} \text{do}\ h(e) : [\,\tau\,]_{\overline{\tau}}}\ [\textsc{Do}]$$

Capability Typing. $\boxed{\Theta \mid \Gamma \vdash_{\text{cap}} h : [\,\mathbb{F}\,]_{\overline{\tau}}}$

$$\frac{\Theta \mid \Gamma \vdash_{\text{cap}} h : [\,\mathbb{F}\,]_{\overline{\tau}}}{\Theta \mid \Gamma \vdash_{\text{cap}} \text{lift}\ h : [\,\mathbb{F}\,]_{\overline{\tau},\,\tau}}\ [\textsc{Cap-Lift}] \qquad \frac{\Theta(c) = [\,\mathbb{F}\,]_{\overline{\tau}}}{\Theta \mid \Gamma \vdash_{\text{cap}} c : [\,\mathbb{F}\,]_{\overline{\tau}}}\ [\textsc{Cap-Var}]$$

$$\frac{\Theta,\ k : [\,\text{Resume}_i\,]_{\overline{\tau}} \mid \Gamma,\ x : \tau' \vdash_{\text{stm}} s : [\,\tau\,]_{\overline{\tau}} \qquad \Sigma(\mathbb{F}) = \tau' \to \tau'' \qquad \Sigma(\text{Resume}_i) = \tau'' \to \tau \qquad i\ \text{fresh}}{\Theta \mid \Gamma \vdash_{\text{cap}} \mathbb{F}(x,\ k) \Rightarrow s : [\,\mathbb{F}\,]_{\overline{\tau},\,\tau}}\ [\textsc{Cap-Handler}]$$

Fig. 3. Typing rules for λ_{Cap}.

The three most interesting rules are Handle, Cap-Lift, and Cap-Handler. They require some detailed explanation.

Handlers introduce delimiters for the continuations, captured by the effect operation they handle. This becomes visible in the rule Handle. While in the conclusion, we type a statement handle $c = h$ in s against a stack shape $\overline{\tau}$, the premises can assume a larger stack shape $\overline{\tau}, \tau$. By installing the delimiter, statement s can safely use the capability c, which has additional control effects at the answer type τ. To guarantee answer type safety, the return type τ of the delimited statement s and the innermost answer type of the larger stack shape $\overline{\tau}, \tau$ have to agree.

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

$$\text{handle}\ c_1 = h_1\ \text{in handle}\ c_2 = h_2\ \text{in do}\ c_1(x)$$

We bind a capability variable c_1 at an outer handler, but want to use it inside of a nested inner handler. While using the capability within the inner handler would be safe, the stack shapes do not match up. To account for this, we allow explicit lifting of capabilities with lift h. In the example, we could thus invoke do (lift c_1)(x). Rule Cap-Lift adjusts a capability h typed against $[\,\mathbb{F}\,]_{\overline{\tau}}$ to be compatible with a larger stack shape $[\,\mathbb{F}\,]_{\overline{\tau},\,\tau}$. This is in spirit similar to *adaptors* in the language Frank [10], to *lift* in Helium [4], and to *inject* in Koka [34]. However, instead of adjusting arbitrary effectful expressions, we only perform the adjustments on capabilities. As we will see, this allows us to guarantee that the lifting itself is performed at compile time.

Finally, rule Cap-Handler checks the body of a handler implementation $\mathbb{F}\ (x,\ k) \Rightarrow s$ against a stack shape $\overline{\tau},\ \tau$. A handler implementation for an effect operation $\mathbb{F}$ takes an argument of type τ' and a continuation k, which can be thought of as an effectful function $\tau'' \to [\ \tau\ ]_{\overline{\tau}}$. Since capabilities will be inlined, we model resumptions as effect operations to guarantee full elimination of the handler abstraction at compile time. The body s of the handler is evaluated in stack shape $\overline{\tau}$, that is, outside of the delimiter that introduced it.

4 TRANSLATION OF λ_{Cap}

In this section, we describe the semantics in terms of a translation to simply-typed lambda calculus [1], extended with a standard **letrec** operator to express λ_{Cap}'s fix. We first describe a translation of λ_{Cap} into iterated CPS where capabilities are still present at runtime. In a second translation, we use staging annotations to prevent administrative redexes and to eliminate capabilities during translation.

4.1 Translating λ_{Cap} to STLC

Figure 4 defines the translation on types and mutually recursive translations of the different syntactic categories of terms. We extend the translation of Schuster and Brachthäuser [49] to the setting of effect handlers. At its heart, our translation is thus an iterated CPS translation [12] but building on the control operator $shift_0$ [37] rather than *shift*, because it more closely fits effect handlers [20, 22, 24]. In Theorem 5.1, we show that our translation takes well-typed λ_{Cap} programs to well-typed STLC programs.

4.1.1 Target Language. The target of our translation is a call-by-value STLC extended with letrec, base types, and primitive operations. As usual we write lambda abstraction as $\lambda x \Rightarrow e$, but use the infix notation $e\ @\ e'$ for application [39]. We sometimes use **let** bindings in the target language assuming the standard shorthand: $\textbf{let}\ \mathsf{x}\ =\ \mathsf{e}\ \textbf{in}\ \mathsf{e}'\ \doteq\ (\lambda \mathsf{x} \Rightarrow \mathsf{e}')\ @\ \mathsf{e}$.

4.1.2 Translation of Types. The translation of types $\mathcal{T}[\![\ \cdot\ ]\!]$ maps base types to base types in STLC and effectful function types $\tau \to [\ \tau'\ ]_{\overline{\tau}}$ to functions from τ to effectful computations $C[\![\ \tau'\ ]\!]_{\overline{\tau}}$. Capability function types are translated to function types in the target language, where the argument type $[\ \mathbb{F}\ ]_{\overline{\tau}}$ (for $\Sigma(\mathbb{F})\ =\ \tau \to \tau'$) is translated like an effectful function type $\tau \to [\ \tau'\ ]_{\overline{\tau}}$.

The meta function $C[\![\ \tau\ ]\!]_{\overline{\tau}}$ computes the type in STLC corresponding to an effectful computation with return type τ in stack shape $\overline{\tau}$. Programs in an empty stack cannot use any control effects and consequently are not CPS translated. The translation of non-empty stack shapes recursively translates the rest of the stack shape. It adds one layer of CPS translation with this recursively translated type as answer type. For example, we have the following translations:

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

From the types, we can see that our translation performs a CPS transformation for each entry in the stack shape $\overline{\tau}$.

4.1.3 Translation of Expressions. In the translation of expressions, we map capability abstraction to ordinary function abstraction and capability application to ordinary function application. The translation of function abstraction and capability application is type directed: the stack shape $\overline{\tau}$ guides the translation of the function body and the handler, respectively.

Translation of Types:

$$
\begin{array}{lcl}
\mathcal{T}[\![\ \mathsf{Int}\ ]\!] & = & \mathsf{Int} \\
\mathcal{T}[\![\ \tau \to [\ \tau'\ ]_{\overline{\tau}}\ ]\!] & = & \mathcal{T}[\![\ \tau\ ]\!] \to C[\![\ \tau'\ ]\!]_{\overline{\tau}} \\
\mathcal{T}[\![\ [\ \mathbb{F}\ ]_{\overline{\tau}} \to \sigma\ ]\!] & = & \mathcal{T}[\![\ [\ \mathbb{F}\ ]_{\overline{\tau}}\ ]\!] \to \mathcal{T}[\![\ \sigma\ ]\!] \\
\mathcal{T}[\![\ [\ \mathbb{F}\ ]_{\overline{\tau}}\ ]\!] & = & \mathcal{T}[\![\ \tau\ ]\!] \to C[\![\ \tau'\ ]\!]_{\overline{\tau}} \\
\quad \text{where} \quad \Sigma(\mathbb{F}) = \tau \to \tau' & & \\
C[\![\ \tau\ ]\!]_{\emptyset} & = & \mathcal{T}[\![\ \tau\ ]\!] \\
C[\![\ \tau\ ]\!]_{\overline{\tau},\ \tau'} & = & (\mathcal{T}[\![\ \tau\ ]\!] \to C[\![\ \tau'\ ]\!]_{\overline{\tau}}) \to C[\![\ \tau'\ ]\!]_{\overline{\tau}}
\end{array}
$$

Translation of Expressions:

$$
\begin{array}{lcl}
\mathcal{E}[\![\ \mathsf{True}\ ]\!] & = & \mathsf{True} \\
\mathcal{E}[\![\ x\ ]\!] & = & x \\
\mathcal{E}[\![\ (x : \tau) \Rightarrow s\ ]\!] & = & \lambda x \Rightarrow \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau}} \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{exp}} (x : \tau) \Rightarrow s : \tau \to [\ \tau'\ ]_{\overline{\tau}} & & \\
\mathcal{E}[\![\ \mathsf{fix}\, f\, (x : \tau) \Rightarrow s\ ]\!] & = & \textbf{letrec}\, f = (\lambda x \Rightarrow \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau}})\ \textbf{in}\ f \\
\quad \text{where } \Theta \mid \Gamma \vdash_{\mathsf{exp}} \mathsf{fix}\, f\, (x : \tau) \Rightarrow s : \tau \to [\ \tau'\ ]_{\overline{\tau}} & & \\
\mathcal{E}[\![\ [c : [\ \mathbb{F}\ ]_{\overline{\tau}}] \Rightarrow e\ ]\!] & = & \lambda c \Rightarrow \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{E}[\![\ e[h]\ ]\!] & = & \mathcal{E}[\![\ e\ ]\!]\ @\ \mathcal{H}[\![\ h\ ]\!]_{\overline{\tau}} \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{cap}} h : [\ \mathbb{F}\ ]_{\overline{\tau}} & &
\end{array}
$$

Translation of Statements:

$$
\begin{array}{lcl}
\mathcal{S}[\![\ e(e')\ ]\!]_{\overline{\tau}} & = & \mathcal{E}[\![\ e\ ]\!]\ @\ \mathcal{E}[\![\ e'\ ]\!] \\
\mathcal{S}[\![\ \mathsf{val}\ x \leftarrow s;\ s'\ ]\!]_{\emptyset} & = & \textbf{let}\ x = \mathcal{S}[\![\ s\ ]\!]_{\emptyset}\ \textbf{in}\ \mathcal{S}[\![\ s'\ ]\!]_{\emptyset} \\
\mathcal{S}[\![\ \mathsf{val}\ x \leftarrow s;\ s'\ ]\!]_{\overline{\tau},\ \tau} & = & \\
\quad \lambda k \Rightarrow \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau},\ \tau}\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\ s'\ ]\!]_{\overline{\tau},\ \tau}\ @\ k) & & \\
\mathcal{S}[\![\ \mathsf{return}\ e\ ]\!]_{\emptyset} & = & \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{S}[\![\ \mathsf{return}\ e\ ]\!]_{\overline{\tau},\ \tau} & = & \lambda k \Rightarrow k\ @\ \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{S}[\![\ \mathsf{do}\ h(e)\ ]\!]_{\overline{\tau}} & = & \mathcal{H}[\![\ h\ ]\!]_{\overline{\tau}}\ @\ \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{S}[\![\ \mathsf{handle}\ c = h\ \mathsf{in}\ s\ ]\!]_{\overline{\tau}} & = & \\
\quad \textbf{let}\ c = \mathcal{H}[\![\ h\ ]\!]_{\overline{\tau},\ \tau}\ \textbf{in}\ \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau},\ \tau}\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\ \mathsf{return}\ x\ ]\!]_{\overline{\tau}}) & & \\
\quad\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\mathsf{stm}} \mathsf{handle}\ c = h\ \mathsf{in}\ s : [\ \tau\ ]_{\overline{\tau}} & &
\end{array}
$$

Translation of Capabilities:

$$
\begin{array}{lcl}
\mathcal{H}[\![\ c\ ]\!]_{\overline{\tau}} & = & c \\
\mathcal{H}[\![\ \mathbb{F}(x,\ k) \Rightarrow s\ ]\!]_{\overline{\tau},\ \tau} & = & \lambda x \Rightarrow \lambda k \Rightarrow \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau}} \\
\mathcal{H}[\![\ \mathsf{lift}\ h\ ]\!]_{\emptyset,\ \tau} & = & \lambda x \Rightarrow \lambda k \Rightarrow k\ @\ (\mathcal{H}[\![\ h\ ]\!]_{\emptyset}\ @\ x) \\
\mathcal{H}[\![\ \mathsf{lift}\ h\ ]\!]_{\overline{\tau},\ \tau,\ \tau'} & = & \\
\quad \lambda x \Rightarrow \lambda k \Rightarrow \lambda k' \Rightarrow \mathcal{H}[\![\ h\ ]\!]_{\overline{\tau},\ \tau}\ @\ x\ @\ (\lambda y \Rightarrow k\ @\ y\ @\ k') & &
\end{array}
$$

Fig. 4. Translation of λ_{Cap} to STLC.

4.1.4 Translation of Statements. The translation of statements $\mathcal{S}[\![\ s\ ]\!]_{\overline{\tau}}$ is indexed by a stack shape $\overline{\tau}$. Source statements s with return type τ in stack shape $\overline{\tau}$ are translated to effectful computations of type $C[\![\ \tau\ ]\!]_{\overline{\tau}}$. In the case of a pure statement without effects, the stack shape is empty and we do not perform a CPS translation. To preserve sharing of results, sequencing translates to a let

binding while returning translates to just the returned expression. In the case where the stack shape is non-empty, we perform a single layer of CPS-translation. We translate the use of a capability (do $h(e)$) to a function application. The translation of handle $c = h$ in s binds c to the translation of h in the translated body s. Importantly, it also delimits effects by applying the translated body to the empty continuation.

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

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

$$\mathcal{S}[\![\ \text{handle}\ c\ =\ h\ \text{in val}\ x \leftarrow \text{do}\ c(\text{True});\ \text{return}\ e\ ]\!]_{\emptyset}$$

Assuming an answer type of Int, we obtain:

$$\textbf{let}\ c\ =\ \mathcal{H}[\![\ h\ ]\!]_{\text{Int}}\ \textbf{in}\quad \overbrace{\mathcal{S}[\![\ \text{val}\ x \leftarrow \text{do}\ c(\text{True});\ \text{return}\ e\ ]\!]_{\text{Int}}\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\ \text{return}\ x\ ]\!]_{\emptyset})}^{\lambda k \Rightarrow \mathcal{S}[\![\ \text{do}\ c(\text{True})\ ]\!]_{\text{Int}}\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\ \text{return}\ e\ ]\!]_{\text{Int}}\ @\ k)}$$

$$\overbrace{c\ @\ \text{True}}^{\mathcal{S}[\![\ \text{do}\ c(\text{True})\ ]\!]_{\text{Int}}} \qquad \overbrace{\lambda k' \Rightarrow k'\ @\ \mathcal{E}[\![\ e\ ]\!]}^{\mathcal{S}[\![\ \text{return}\ e\ ]\!]_{\text{Int}}\ @\ k}$$

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

$$\textbf{let}\ c\ =\ \mathcal{H}[\![\ h\ ]\!]_{\text{Int}}\ \textbf{in}\quad \lambda k \Rightarrow c\ @\ \text{True}\ @\ (\lambda x \Rightarrow (\lambda k' \Rightarrow k'\ @\ \mathcal{E}[\![\ e\ ]\!])\ @\ k)\ @\ (\lambda x \Rightarrow x)$$

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

4.2 Removing Handler Abstractions by Staging

The program resulting from translating the above example still abstracts over the handler implementation and exhibits several administrative redexes. In this section we switch our target language to 2-level lambda calculus [23, 39, 50] to guarantee that certain redexes never occur in the generated program. Staging annotations are automatically inserted as part of the definition of our translation.

There are two classes of redexes that we want to avoid in the generated program. Firstly, we use staging to avoid generating administrative beta redexes in our CPS translation. This standard use of staging in the translation of control operators has been introduced by Danvy and Filinski [13]. We build on a variant, which is generalized to the setting of iterated CPS [22, 49]. Secondly, we use staging to avoid generating redexes associated with the effect handler abstraction. That is, handling

effects, calling effect operations and lifting capabilities should not introduce any additional redexes in the generated program. Figure 5 shows our translation to 2-level lambda calculus.

2-level lambda calculus. The general idea of staging is to mark some abstractions and applications as static and some as residual. Static redexes will be reduced during translation, while residual redexes will be generated, that is, residualized. We use standard notation that we briefly review. On the type level we use red color and an underline for types of *residual* terms (i.e. terms that will be residualized). For example $\underline{\mathsf{Int}} \to \underline{\mathsf{Int}}$ is the type of a generated function from integers to integers. We write types of *static* (i.e. stage time) terms in blue with an overbar. For example $\underline{\mathsf{Int}} \mathrel{\overline{\to}} \underline{\mathsf{Int}}$ is the type of a static function between residualized integers. Similarly, on the term level we write residual terms in red with an underscore. For example, $\underline{1} \mathbin{\underline{+}} \underline{2}$ is the term that adds the integer one and the integer two. This redex will occur in the generated program. We write terms that we evaluate during translation in blue with an overbar. For example $(\overline{\lambda x \Rightarrow x}) \mathbin{\overline{@}} \underline{5}$ will statically evaluate to the term $\underline{5}$.

We use $\underline{C}[\![\ \tau\ ]\!]_{\overline{\tau}}$ to describe the type of residual effectful computations. The whole computation type $C[\![\ \tau\ ]\!]_{\overline{\tau}}$ as defined in Figure 4 is residualized. The type $\overline{C}[\![\ \tau\ ]\!]_{\overline{\tau}}$ represents static computations. Importantly, while the answer types are residual (e.g., $\underline{\tau}$) the structure of the computation is static.

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

$$
\begin{aligned}
&\textsc{Reify}_{\overline{\tau}} \quad : \overline{C}[\![\ \tau\ ]\!]_{\overline{\tau}} \to \underline{C}[\![\ \tau\ ]\!]_{\overline{\tau}} \\
&\textsc{Reify}_{\emptyset}\ s \doteq s \\
&\textsc{Reify}_{\overline{\tau},\tau}\ s \doteq \underline{\lambda k \Rightarrow}\ \textsc{Reify}_{\overline{\tau}}\ (s\ \overline{@}\ (\overline{\lambda x \Rightarrow}\ \textsc{Reflect}_{\overline{\tau}}\ (k\ \underline{@}\ x)))
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{Reflect}_{\overline{\tau}} \quad : \underline{C}[\![\ \tau\ ]\!]_{\overline{\tau}} \to \overline{C}[\![\ \tau\ ]\!]_{\overline{\tau}} \\
&\textsc{Reflect}_{\emptyset}\ s \doteq s \\
&\textsc{Reflect}_{\overline{\tau},\tau}\ s \doteq \overline{\lambda k \Rightarrow}\ \textsc{Reflect}_{\overline{\tau}}\ (s\ \underline{@}\ (\underline{\lambda x \Rightarrow}\ \textsc{Reify}_{\overline{\tau}}\ (k\ \overline{@}\ x)))
\end{aligned}
$$

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

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

4.2.3 Statements. We translate statements typed against $[\ \tau\ ]_{\overline{\tau}}$ to *static* computations of type $\overline{C}[\![\ \tau\ ]\!]_{\overline{\tau}}$. We want to preserve function applications, so we generate an application and reflect the resulting statement. To preserve sharing, we translate sequenced pure statements to a residual let binding. When translating sequencing and returning of effectful statements, we mark all continuation abstractions and applications as static. This allows us to avoid administrative beta-redexes [13]. We translate the binding of capability variables in handlers and the use of capabilities to static binding and application. This ensures that capabilities are fully inlined at their call-site.

Translation of Types:

$$
\begin{array}{lcl}
\mathcal{T}[\![\ \mathsf{Int}\ ]\!] & = & \underline{\mathsf{Int}} \\
\mathcal{T}[\![\ \tau \to [\ \tau'\ ]_{\overline{\tau}}\ ]\!] & = & \mathcal{T}[\![\ \tau\ ]\!] \underline{\to} \underline{C}[\![\ \tau'\ ]\!]_{\overline{\tau}} \\
\mathcal{T}[\![\ [\ \mathbb{F}\ ]_{\overline{\tau}} \to \sigma\ ]\!] & = & \mathcal{T}[\![\ [\ \mathbb{F}\ ]_{\overline{\tau}}\ ]\!] \overline{\to} \mathcal{T}[\![\ \sigma\ ]\!] \\
\mathcal{T}[\![\ [\ \mathbb{F}\ ]_{\overline{\tau}}\ ]\!] & = & \mathcal{T}[\![\ \tau\ ]\!] \overline{\to} \overline{C}[\![\ \tau'\ ]\!]_{\overline{\tau}} \\
\quad \text{where} \quad \Sigma(\mathbb{F}) = \tau \to \tau' & & \\
\underline{C}[\![\ \tau\ ]\!]_{\emptyset} & = & \mathcal{T}[\![\ \tau\ ]\!] \\
\underline{C}[\![\ \tau\ ]\!]_{\overline{\tau},\,\tau'} & = & (\mathcal{T}[\![\ \tau\ ]\!] \underline{\to} \underline{C}[\![\ \tau'\ ]\!]_{\overline{\tau}}) \underline{\to} \underline{C}[\![\ \tau'\ ]\!]_{\overline{\tau}} \\
\overline{C}[\![\ \tau\ ]\!]_{\emptyset} & = & \mathcal{T}[\![\ \tau\ ]\!] \\
\overline{C}[\![\ \tau\ ]\!]_{\overline{\tau},\,\tau'} & = & (\mathcal{T}[\![\ \tau\ ]\!] \overline{\to} \overline{C}[\![\ \tau'\ ]\!]_{\overline{\tau}}) \overline{\to} \overline{C}[\![\ \tau'\ ]\!]_{\overline{\tau}}
\end{array}
$$

Translation of Expressions:

$$
\begin{array}{lcl}
\mathcal{E}[\![\ \mathsf{True}\ ]\!] & = & \underline{\mathsf{True}} \\
\mathcal{E}[\![\ x\ ]\!] & = & x \\
\mathcal{E}[\![\ (x : \tau) \Rightarrow s\ ]\!] & = & \underline{\lambda x \Rightarrow}\ \textsc{Reify}_{\overline{\tau}}\ \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau}} \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\text{exp}} (x : \tau) \Rightarrow s : \tau \to [\ \tau'\ ]_{\overline{\tau}} & & \\
\mathcal{E}[\![\ \mathsf{fix}\, f\, (x : \tau) \Rightarrow s\ ]\!] & = & \\
\quad \underline{\mathsf{letrec}}\, f = (\underline{\lambda x \Rightarrow}\ \textsc{Reify}_{\overline{\tau}}\ \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau}})\ \underline{\mathsf{in}}\ f & & \\
\quad \text{where } \Theta \mid \Gamma \vdash_{\text{exp}} \mathsf{fix}\, f\, (x : \tau) \Rightarrow s : \tau \to [\ \tau'\ ]_{\overline{\tau}} & & \\
\mathcal{E}[\![\ [c : [\ \mathbb{F}\ ]_{\overline{\tau}}] \Rightarrow e\ ]\!] & = & \overline{\lambda c \Rightarrow}\ \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{E}[\![\ e[h]\ ]\!] & = & \mathcal{E}[\![\ e\ ]\!]\ \overline{@}\ \mathcal{H}[\![\ h\ ]\!]_{\overline{\tau}} \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\text{cap}} h : [\ \mathbb{F}\ ]_{\overline{\tau}} & &
\end{array}
$$

Translation of Statements:

$$
\begin{array}{lcl}
\mathcal{S}[\![\ e(e')\ ]\!]_{\overline{\tau}} & = & \textsc{Reflect}_{\overline{\tau}}\ (\mathcal{E}[\![\ e\ ]\!]\ \underline{@}\ \mathcal{E}[\![\ e'\ ]\!]) \\
\mathcal{S}[\![\ \mathsf{val}\, x \leftarrow s;\ s'\ ]\!]_{\emptyset} & = & \underline{\mathsf{let}}\ x = \mathcal{S}[\![\ s\ ]\!]_{\emptyset}\ \ \underline{\mathsf{in}}\ \mathcal{S}[\![\ s'\ ]\!]_{\emptyset} \\
\mathcal{S}[\![\ \mathsf{val}\, x \leftarrow s;\ s'\ ]\!]_{\overline{\tau},\,\tau} & = & \\
\quad \overline{\lambda k \Rightarrow}\ \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau},\,\tau}\ \overline{@}\ (\overline{\lambda x \Rightarrow}\ \mathcal{S}[\![\ s'\ ]\!]_{\overline{\tau},\,\tau}\ \overline{@}\ k) & & \\
\mathcal{S}[\![\ \mathsf{return}\ e\ ]\!]_{\emptyset} & = & \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{S}[\![\ \mathsf{return}\ e\ ]\!]_{\overline{\tau},\,\tau} & = & \overline{\lambda k \Rightarrow}\ k\ \overline{@}\ \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{S}[\![\ \mathsf{do}\ h(e)\ ]\!]_{\overline{\tau}} & = & \mathcal{H}[\![\ h\ ]\!]\ \overline{@}\ \mathcal{E}[\![\ e\ ]\!] \\
\mathcal{S}[\![\ \mathsf{handle}\ c = h\ \mathsf{in}\ s\ ]\!]_{\overline{\tau}} & = & \\
\quad \overline{\mathsf{let}}\ c = \mathcal{H}[\![\ h\ ]\!]_{\overline{\tau},\,\tau}\ \overline{\mathsf{in}}\ \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau},\,\tau}\ \overline{@}\ (\overline{\lambda x \Rightarrow}\ \mathcal{S}[\![\ \mathsf{return}\ x\ ]\!]_{\overline{\tau}}) & & \\
\quad \text{where} \quad \Theta \mid \Gamma \vdash_{\text{stm}} \mathsf{handle}\ c = h\ \mathsf{in}\ s : [\ \tau\ ]_{\overline{\tau}} & &
\end{array}
$$

Translation of Capabilities:

$$
\begin{array}{lcl}
\mathcal{H}[\![\ c\ ]\!]_{\overline{\tau}} & = & c \\
\mathcal{H}[\![\mathbb{F}(x,\ k) \Rightarrow s\ ]\!]_{\overline{\tau},\,\tau} & = & \overline{\lambda x \Rightarrow}\ \overline{\lambda k \Rightarrow}\ \mathcal{S}[\![\ s\ ]\!]_{\overline{\tau}} \\
\mathcal{H}[\![\ \mathsf{lift}\ h\ ]\!]_{\emptyset,\,\tau} & = & \overline{\lambda x \Rightarrow}\ \overline{\lambda k \Rightarrow}\ k\ \overline{@}\ (\mathcal{H}[\![\ h\ ]\!]_{\emptyset}\ \overline{@}\ x) \\
\mathcal{H}[\![\ \mathsf{lift}\ h\ ]\!]_{\overline{\tau},\,\tau,\,\tau'} & = & \\
\quad \overline{\lambda x \Rightarrow}\ \overline{\lambda k \Rightarrow}\ \overline{\lambda k' \Rightarrow}\ \mathcal{H}[\![\ h\ ]\!]_{\overline{\tau}}\ \overline{@}\ x\ \overline{@}\ (\overline{\lambda y \Rightarrow}\ k\ \overline{@}\ y\ \overline{@}\ k') & &
\end{array}
$$

Fig. 5. Translation of λ_{Cap} to 2-level lambda calculus.

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

Example. Applying the translation to 2-level lambda calculus to the example from the previous subsection, we obtain

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

which reduces statically to:

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

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

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

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

to the following static capability abstraction:

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

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

5 EVALUATION

We implemented λ_{Cap} as a *shallow embedding* into the dependently typed programming language Idris [7]. We use typed HOAS [41] and represent the AST of residualized programs of type $\underline{\tau}$ as values of a data type indexed by the type τ. We use the host language Idris to both express source programs, as well as to express static abstractions and applications. Throughout our implementation, we use dependent types to index source and target programs by their types, including stack shapes, which we represent as a type-level list of types. Our translation follows the inductive structure of this type-level list.

5.1 Theoretical Results

Our translation satisfies a few meta-theoretic properties. In our implementation, we were careful to make these properties hold by construction. Firstly, our unstaged translation (as presented in Figure 4) preserves well-typedness.

Theorem 5.1 (Typability of translated terms – unstaged).

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

Proof. By induction over the typing derivations and case distinction on the stack shapes – see Appendix C.1. □

Importantly, we obtain effect safety as a corollary.

Corollary 5.2 (Effect safety). *Given a closed statements s, if* $\emptyset \mid \emptyset \vdash s : [\![\ \tau\ ]\!]_{\emptyset}$*, then evaluating* $\mathcal{S}[\![\ s\ ]\!]_{\emptyset}$ *will not get stuck.*

Effect safety immediately follows from Theorem 5.1 and soundness of STLC. Well-typedness is also preserved by the staged translation (Figure 5).

Theorem 5.3 (Typability of translated terms – staged).

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

Proof. By induction over the typing derivations and case distinction on the stack shapes – see Appendix C.2. □

That is, our translation takes well-typed source programs to well-typed 2-level lambda calculus programs. From Theorem 5.3 and soundness of the 2-level lambda calculus follows *stage time correctness*: our translation only applies static functions statically and residualizes applications of residual functions. In our implementation, we ensure this by distinguishing static and residual expressions on the type-level. Stage time correctness means that code generation does not fail for well-typed programs:

Theorem 5.4 (Full residualization). *Given a closed statement s, if* $\emptyset \mid \emptyset \vdash s : [\![\ \tau\ ]\!]_{\emptyset}$ *then its translation* $\mathcal{S}[\![\ s\ ]\!]_{\emptyset}$ *can be fully reduced to a residualized term.*

Proof. By Theorem 5.3, we have that $\vdash \mathcal{S}[\![\ s\ ]\!]_{\emptyset} : \overline{C}[\![\ \tau\ ]\!]_{\emptyset}$. We can compute $\overline{C}[\![\ \tau\ ]\!]_{\emptyset} = \mathcal{T}[\![\ \tau\ ]\!]$. By induction on the rules of $\mathcal{T}[\![\ \cdot\ ]\!]$, we get that translation of dynamic types τ results in some residual type $\underline{\tau'}$. Soundness of the 2-level lambda calculus and the fact that our translation does not introduce any static **letrec** finally guarantees that stage-time reduction will not get stuck. □

By Corollary 5.4 and soundness of the 2-level lambda calculus, it is easy to see that effect safety (Corollary 5.2) also extends to the staged translation. That is, reducing the residualized term will not get stuck.

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

Theorem 5.5 (Zero overhead). *The translations of capability abstraction, capability application,* do $h(e)$, handle $c = h$ in s, $\mathbb{F}(x, k) \Rightarrow s$, *and* lift h *do not introduce any residual lambda abstractions or applications, except for those in the translation of their subterms.*

Proof. By inspection of our translation with staging annotations in Figure 5. All abstractions and applications that the translations immediately introduce are marked as static. □

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

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

5.2 Performance Results

We now assess how well our translated code performs relative to existing languages with effect handlers. Implementing those benchmarks also provides evidence that we can express typical programs using effect handlers in λ_{Cap} despite its restrictions (which we discuss in Section 6). The results are shown in Figure 6. All benchmarks were executed on a 2.60GHz Intel(R) Core(TM) i7 with 11GB of RAM.

We compare our implementation with Koka (0.9.0) [33], Multicore OCaml (4.06.1) [16], and an implementation of delimited control operators in Chez Scheme (9.5.3) [27]. For each comparison, we generate code in CPS in the corresponding language (that is, JavaScript, OCaml, and Scheme) and make sure to use the same primitive functions and data structures that the baseline uses. For each of the example functions we generate code in two flavors that we call `Unstaged` (Figure 4) and `Staged` (Figure 5). In our implementation, we additionally apply standard techniques [13, 49] to avoid generating administrative eta redexes. We report the mean and standard deviation of the runtime of the programs under consideration.

We report numbers for three example programs. The Triple program is inspired by the example in Danvy and Filinski [12]. It uses the running examples choice from Section 2 to find triples of numbers that sum up to a given target number. The Queens example places queens on a chess board and is taken from Kiselyov and Sivaramakrishnan [30]. The Count benchmark appears in Kiselyov and Ishii [28] and Wu and Schrijvers [52] and counts down recursively using a single state effect.

The benchmark results are generally encouraging. They indicate that the code we generate with and without staging is significantly faster than in the languages we compare against with a speedup of 102x, 21x, and 13x respectively for the Triple benchmark and a speedup of 256x, 10x, and 54x respectively for the Count benchmark. Both benchmarks extensively use control effects and yield optimization opportunities across effect operations for us to exploit. In the other benchmarks we generally observe speedups as well. Interestingly, in the Queens benchmark we do not observe any speedup between our unstaged translation and our staged translation. It uses a single effect operation in a single place and handles it as a loop which our staged translation immediately residualizes. We now discuss the setup and observations specific to each of the baselines, we compare against.

	Time in ms (Standard Deviation)		
Benchmark	**Baseline**	**Unstaged λ_{Cap}**	**Staged λ_{Cap}**
Koka			
Triple	2367.1 ±27.0	62.3 ±2.0	23.1 ±0.5
Queens (18)	382.7 ±5.5	167.0 ±1.3	169.0 ±1.6
Count (2K)	49.4 ±0.5	0.4 ±0.0	0.2 ±0.0
Multicore OCaml			
Triple	49.9 ±0.7	4.5 ±0.1	2.3 ±0.1
Queens (18)	53.8 ±0.3	32.1 ±0.5	32.9 ±0.5
Count (1M)	72.6 ±0.7	19.6 ±3.3	7.4 ±0.2
Primes (1K)	32.3 ±0.6	28.9 ±0.6	22.7 ±0.2
Chameneos	45.3 ±0.8	31.9 ±0.9	28.6 ±0.8
Chez Scheme			
Triple	47.8 ±0.8	3.7 ±0.0	3.7 ±0.0
Queens (18)	233.2 ±1.5	230.3 ±1.0	229.9 ±0.9
Count (1M)	476.2 ±4.5	8.5 ±0.4	8.8 ±0.4

Fig. 6. Comparing the performance of λ_{Cap} with Koka, Multicore OCaml, and Chez Scheme.

Comparison with Koka. Koka compiles to JavaScript and uses a standard library of functions and data types also compiled to JavaScript. In our comparison with Koka, we do not generate Koka but JavaScript code in CPS using the same standard library where possible. Benchmarks were executed using the JavaScript library `benchmark.js`[2] on `Node.js`[3] version 12.11.1. For the Count benchmark, we had to use a smaller initial state than in the other comparisons because the code generated by Koka as well as the code generated by our translation leads to a stack overflow for larger numbers. Koka already performs a selective CPS transformation. However, removing the runtime search for handler implementations causes significant speedups.

Comparison with Multicore OCaml. Multicore [16] is a fork of the OCaml compiler [35] that adds support for effect handlers. We compile the Multicore OCaml programs with the multicore variant and our generated code with the standard variant of the `ocamplopt` compiler (4.06.1). Each program is compiled to a standalone executable and we measure the running time with the `bench` program[4]. In our comparison with Multicore OCaml, we benchmarked two additional examples from an online repository of Multicore OCaml examples[5]: Chameneos and Primes. These two benchmarks exercise the use-case that Multicore OCaml was designed for, that is, resuming continuations only once. While our translation always supports resuming the continuation multiple times, even in these special cases we observe slight speedups. The code uses native side effects like for example mutating a global queue which we make execute in the right order by inserting let bindings as part of our translation.

Comparison with DelimCC on Chez Scheme. We also assess the performance of our generated code relative to a fast implementation of delimited continuations without any effect handling code. For this comparison, we implemented the examples using ordinary functions that capture the current continuation via `shift0` [11]. We use the DelimCC library [27] and compile the example programs as well as our generated code with the Chez Scheme compiler [18]. We make two observations

[2]https://benchmarkjs.com/
[3]https://nodejs.org
[4]http://hackage.haskell.org/package/bench
[5]https://github.com/kayceesrk/effects-examples

on the benchmark results. In the Queens benchmark our generated code exhibits about the same runtime as the baseline code using `shift0`. Furthermore, in all three benchmarks we do not observe any speedup of the code generated using staging over the code generated without any staging. We conjecture that the reason for the second observation is that our generated code (even without staging) exposes static beta reductions that an optimizing compiler like the Chez Scheme compiler will perform at compile time. Using staging annotations we *guarantee* that these reductions happen at compile time.

6 LIMITATIONS AND FUTURE WORK

In this section, we discuss the restrictions we imposed on λ_{Cap} in order go guarantee the full elimination of abstractions and applications related to handlers. We characterize the kind of programs ruled out by these restrictions and how and at what cost they might be supported in the future.

Stack shapes are monomorphic. Our translation assumes all stack shapes to be fully known. We can still express most programs in λ_{Cap} by *monomorphization*. For each function and stack shape, we can generate a specialized version that takes the appropriate numbers of continuations. Monomorphization leads to an increase in the size of the generated code and rules out separate compilation. Instead, we could add support for *stack shape polymorphism*, which would allow us to translate stack shape polymorphic functions once and then use them in different contexts at different stack shapes. Monomorphization also rules out recursive functions where the recursive call occurs under a new handler:

```
def loop(n) { handle fail = Fail((), k) ⇒ () in loop(n − 1) }
```

We could say that examples like this use *stack-shape polymorphic recursion*. It is unclear to us how common these kinds of programs are in practice. Compiling stack-shape polymorphic functions, requires a uniform representation for an arbitrary number of continuations, for example a list [19, 22]. Such a representation, would require a runtime lookup to find the correct subcontinuation. This has a run-time cost and, more central to this work, blocks compile-time optimizations.

Capabilities are second class. Capabilities in λ_{Cap} are *second class* [40]. This allows us to prove that they never appear in translated programs, but prevents us from writing certain kinds of programs in λ_{Cap}. For example, a common idiom in Koka is to abstract over handlers like in the following example:

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

However, this example is ruled out by our type system. Being a parameter, prog has a dynamic type, but capability application prog[fail] demands that prog has a static type. We could easily lift this restriction at the cost of losing the inlining guarantees for handler implementations as per Theorem 5.5.

Handlers are statically scoped. Traditionally, effect operations are handled by the *dynamically* closest enclosing effect handler [2]. Like Brachthäuser and Schuster [5] and Zhang and Myers [53], in λ_{Cap} we employ explicit capability passing, which means that effect operations are statically scoped. This way, λ_{Cap} cannot express state handlers in the style of Biernacki, Piróg, Polesiuk, and Sieczkowski [3], which are expressible in Koka as:

```
fun handlePut(prog) {
  handle(prog) {
    put(s) -> inject<get>({handle({resume(())}) {
      get() -> resume(s)
    }})
  }
}
```

Again, we could support dynamically scoped handler implementations at the cost of losing inlining guarantees for those handler implementations. It is not clear to us, how severe this restriction is in practice. To emulate dynamically bound effect operations, in the future, we would like to integrate delimited dynamic binding [29] into λ_{Cap}.

Generated code is in CPS. We generate code in CPS, which can be disadvantageous for some languages or virtual machines. Targeting a language with support for the delimited control operator $shift_0$, we could instead use $shift_0$ directly instead of translating to iterated CPS. However, implementing delimited control in terms of iterated CPS is crucial to achieve compile-time optimization. It allows us to make use of the static knowledge of the context around the invocation of effect operations. However, we see potential for improvement in the future. It is common practice to compile to CPS [26], an explicit representation of join points [38], or both [9]. In the future, we want to target an intermediate language with an explicit representation of continuations and treat continuations differently from functions at compile time and run time. For example, we could extend the intermediate language presented in [26] generalizing from two continuations to an arbitrary number.

7 RELATED WORK

We combine capability passing with an implementation of control effects by iterated CPS transformation. This unique combination allows us to exploit static information and to eliminate all overhead introduced by the effect handler abstraction. In this section we relate our approach to existing work on capability passing, on compilation of effect handlers via CPS transformation, and on optimization of programs using effect handlers.

7.1 Capability Passing

Zhang and Myers [53] present a language $\lambda_{\Downarrow\Uparrow}$ that features *abstraction safety* and demonstrate modular reasoning about effect-polymorphic higher-order functions. λ_{Cap} is modeled after $\lambda_{\Downarrow\Uparrow}$ and inherits the static binding of effect handlers as explained in Section 6. An important difference is in our treatment of effect types. Their effect types are sets of labels, where each label stands for an occurrence of a delimiter in the program. In contrast, our effect types are *ordered lists* of types, where each type is the answer type at an enclosing delimiter. Their primary goal is modular reasoning over effect-parametric functions, while our goal is exploiting static information for efficient compilation.

Brachthäuser et al. [6] present an implementation of effect handlers in explicit capability-passing style in Java. They implement continuation capture by a translation of JVM bytecode to CPS and an implementation of multi-prompt delimited continuations [19]. Like our approach, their capabilities contain handler implementations, but they also contain a prompt to enable capturing the part of the stack up to the corresponding delimiter. Passing the capabilities explicitly facilitates optimizations by the JVM. We make this heuristic explicit and guarantee inlining of handler implementations.

Kammar et al. [24] present multiple translations of effect handlers into Haskell. They translate handler implementations to type class instances, turning handlers into dictionary parameters of effectful functions. This can be seen as capability passing. Furthermore, they present a translation that uses nested applications of the continuation monad for multiple handlers. This translation is very similar to the translation of λ_{Cap} to iterated CPS that we present here.

7.2 Implementing Control-Effects by CPS Translation

Hillerström et al. [22] present an implementation technique for effect handlers that can be seen as an iterated CPS transformation. An important difference is that their source language supports dynamically bound handler implementations, while we support statically bound handlers via capability passing. Therefore in their translation of handlers, each handler matches on the effect operation at *run time* to decide whether it should handle it or forward it to an outer handler. In contrast, we explicitly pass handler implementations, which allows us to guarantee full inlining. This comes at the cost of not supporting dynamically scoped handlers, as discussed in Section 6. They report their curried CPS translation to be a composition of an implementation of effect handlers [20] and an implementations of delimited continuations in terms of a CPS transformation [37]. Similarly, our type system and translation into iterated CPS is also close to the one by Materzok and Biernacki [37]. However, we simplify the type system by not supporting answer type modification, which makes our stack shapes lists rather than trees.

Leijen [33] compiles algebraic effects by CPS transformation. For performance the translation is *selective*, distinguishing between pure and effectful parts of the program on the type level. Guided by the types, only effectful parts of a program are CPS translated. Our work can be seen as a generalization in that we do not only distinguish pure and effectful parts of the program, but track the *number* of control effects in the type system. Guided by these types, we translate different parts of the program to work with different numbers of continuations arguments.

We follow Schuster and Brachthäuser [49] and represent stack shapes as list of answer types to drive a type-directed translation into the CPS hierarchy. While their goal is to efficiently implement control operators, our goal is the elimination of overhead introduced by effect handlers.

7.3 Optimization of effect handlers

Pretnar et al. [46] show how to reduce the overhead incurred by using effect handlers by compile-time optimizations. While their approach is to apply semantics preserving rewrite rules, we translate effect handlers to a 2-level lambda calculus in CPS and apply beta-reductions at compile time. Our approach has the advantage that our optimizations are semantics preserving by construction, while they report their rewritings to be "fragile and have been postponed" [48]. As a downside, our translation might miss optimization opportunities that are specific to effect handlers and only become apparent in a language where they are explicitly represented.

Wu and Schrijvers [52] consider effectful programs as a free monad over a signature of effect operations. They fuse multiple handlers to avoid building and then folding any intermediate free monad structure in memory. They achieve excellent performance on a number of benchmarks, which validates their optimization method. Their optimization crucially relies on inlining and function specialization. Since their implementation uses Haskell and GHC, they use Haskell type classes to trigger function specialization. To get access to the current continuation, they use nested layers of the codensity monad which is the continuation monad with a polymorphic answer type. This is a similarity to our translation to nested layers of CPS.

8 CONCLUSION

We have presented λ_{Cap}, a language with effect handlers in explicit capability-passing style. Its syntax and type system restricts programs to make it possible to always statically know handler implementations and stack shapes. We have given a translation of λ_{Cap} to STLC that exploits this static knowledge to eliminate all overhead introduced by abstracting over effect operations. Since λ_{Cap} exposes a lot of details for the efficient compilation of effect handlers, in the future we will investigate how to translate a more high-level language with effect handlers to λ_{Cap}.

REFERENCES

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

[2] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

[3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2, POPL, Article 8 (Dec. 2017), 30 pages.

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

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

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

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

[8] Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile Event Correlation with Algebraic Effects. *Proc. ACM Program. Lang.* 2, ICFP, Article 67 (July 2018), 31 pages.

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

[10] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2019. *Doo Bee Doo Bee Doo*. Technical Report. The University of Edinburgh. `http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-february2019.pdf`

[11] Olivier Danvy and Andrzej Filinski. 1989. A functional abstraction of typed contexts. *DIKU Rapport 89/12, DIKU, University of Copenhagen* (1989).

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

[13] Oliver Danvy and Andrzej Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391.

[14] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (Nov. 1996), 730–751.

[15] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent system programming with effect handlers. In *Proceedings of the Symposium on Trends in Functional Programming*. Springer LNCS 10788.

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

[17] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective concurrency through algebraic effects. In *OCaml Workshop*.

[18] R. Kent Dybvig. 2006. The Development of Chez Scheme. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 1–12. doi:10.1145/1159803.1159805

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

[20] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.* 1, ICFP, Article 13 (Aug. 2017), 29 pages.

[21] Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the Workshop on Type-Driven Development*. ACM, New York, NY, USA.

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

Informatik.

[23] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA.

[24] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the International Conference on Functional Programming*. ACM, New York, NY, USA, 145–158.

[25] Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming* 27, 1 (Jan. 2017).

[26] Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the International Conference on Functional Programming*. ACM, New York, NY, USA, 177–190.

[27] Oleg Kiselyov. 2009. Multi-prompt delimited control in R5RS Scheme. (Oct. 2009). `http://okmij.org/ftp/continuations/implementations.html#delimcc-scheme`.

[28] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the Haskell Symposium.* ACM, New York, NY, USA, 94–105.

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

[30] Oleg Kiselyov and KC Sivaramakrishnan. 2018. Eff Directly in OCaml. In *Proceedings of the ML Family Workshop / OCaml Users and Developers workshops (Electronic Proceedings in Theoretical Computer Science)*, Kenichi Asai and Mark Shinwell (Eds.), Vol. 285. Open Publishing Association, 23–58. doi:10.4204/EPTCS.285.2

[31] Daan Leijen. 2017a. Implementing Algebraic Effects in C. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer International Publishing, Cham, Switzerland, 339–363.

[32] Daan Leijen. 2017b. Structured Asynchrony with Algebraic Effects. In *Proceedings of the Workshop on Type-Driven Development*. ACM, New York, NY, USA, 16–29.

[33] Daan Leijen. 2017c. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 486–499.

[34] Daan Leijen. 2018. First Class Dynamic Effect Handlers: Or, Polymorphic Heaps with Dynamic Effect Handlers. In *Proceedings of the Workshop on Type-Driven Development*. ACM, New York, NY, USA, 51–64.

[35] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2017. The OCaml system release 4.06. *Institut National de Recherche en Informatique et en Automatique* (2017).

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

[37] Marek Materzok and Dariusz Biernacki. 2012. A dynamic interpretation of the CPS hierarchy. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 296–311.

[38] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling Without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 482–494. doi:10.1145/3062341.3062380

[39] Flemming Nielson and Hanne Riis Nielson. 1996. Multi-level lambda-calculi: an algebraic description. In *Partial evaluation*. Springer, 338–354.

[40] Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. 2016. 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*. ACM, New York, NY, USA, 234–251.

[41] Frank Pfenning and Conal Elliot. 1988. Higher-Order Abstract Syntax. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 199–208. doi:10.1145/53990.54010

[42] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. ACM, New York, NY, USA, 809–818. doi:10.1145/3209108.3209166

[43] Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *European Symposium on Programming*. Springer-Verlag, 80–94.

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

[45] Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.

[46] Matija Pretnar, Amr Hany Shehata Saleh, Axel Faes, and Tom Schrijvers. 2017. *Efficient compilation of algebraic effects and handlers.* Technical Report. Department of Computer Science, KU Leuven; Leuven, Belgium.

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

[48] Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. 2018. Explicit Effect Subtyping. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, Switzerland,

327–354.
[49] Philipp Schuster and Jonathan Immanuel Brachthäuser. 2018. Typing, Representing, and Abstracting Control. In *Proceedings of the Workshop on Type-Driven Development*. ACM, New York, NY, USA, 14–24. doi:10.1145/3240719.3241788
[50] Walid Taha and Tim Sheard. 2000. MetaML and Multi-stage Programming with Explicit Annotations. *Theoretical Computer Science* 248, 1-2 (Oct. 2000), 211–242. http://dx.doi.org/10.1016/S0304-3975(00)00053-0
[51] Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94.
[52] Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free - Efficient Algebraic Effect Handlers. In *Proceedings of the Conference on Mathematics of Program Construction*. Springer LNCS 9129.
[53] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages.

A TARGET LANGUAGE OF THE UNSTAGED TRANSLATION (STLC)

For easier reference, Figure 7 repeats the standard syntax and typing rules of a call-by-value simply typed lambda calculus [1] extended with **letrec**.

B TARGET LANGUAGE OF THE STAGED TRANSLATION 2λ

Figure 8 repeats the standard syntax and typing rules of a 2-level lambda calculus [14, 22, 23, 39]. For simplicity, and to be closer to our implementation, we only include residualized constants and only residualized **letrec**.

C PROOFS

C.1 Typability of Translated Terms (Unstaged)

Our translation preserves typability (Theorem 5.1). To proof this theorem, we have to assume hygiene of our translation. That is, variables in $\mathcal{T}[\![\,\Theta\,]\!]$ and $\mathcal{T}[\![\,\Gamma\,]\!]$ do not shadow each other, and fresh variables in Γ, introduced by the translation (that is, not present in the source term) do not shadow variables in $\mathcal{T}[\![\,\Gamma\,]\!]$. Furthermore, we assume that for every $\mathbb{F}$, there exists a corresponding global signature $\Sigma(\mathbb{F}) = \tau_1 \to \tau_2$.

Proof. The proof proceeds by induction on the typing derivation. The cases for the typing judgement $\vdash_{\text{exp}}$ are all straightforward. The most interesting cased in the typing judgement $\vdash_{\text{cap}}$ are rule Cap-Lift and rule Cap-Handler. We start with rule Cap-Lift, where (following the translation) we need to make a case distinction between the singleton stack shape $(\emptyset, \tau)$ and the stack shape with at least two elements $(\overline{\tau}, \tau, \tau')$.

case Cap-Lift – stack shape $\emptyset, \tau$

Given $\Theta \mid \Gamma \vdash_{\text{cap}} \mathsf{lift}\ h : [\![\,\mathbb{F}\,]\!]_{\emptyset,\tau}$ we need to show $\mathcal{T}[\![\,\Theta\,]\!], \mathcal{T}[\![\,\Gamma\,]\!] \vdash \mathcal{H}[\![\,\mathsf{lift}\ h\,]\!] : \mathcal{T}[\![\,[\![\,\mathbb{F}\,]\!]_{\emptyset,\tau}\,]\!]$. From the premises, we have $\Theta \mid \Gamma \vdash_{\text{cap}} h : [\![\,\mathbb{F}\,]\!]_{\emptyset}$ **(1)**. Further, we can compute: $\mathcal{T}[\![\,[\![\,\mathbb{F}\,]\!]_{\emptyset,\tau}\,]\!] = \mathcal{T}[\![\,\tau_1\,]\!] \to C[\![\,\tau_2\,]\!]_{\emptyset,\tau} = \mathcal{T}[\![\,\tau_1\,]\!] \to (\mathcal{T}[\![\,\tau_2\,]\!] \to \mathcal{T}[\![\,\tau\,]\!]) \to \mathcal{T}[\![\,\tau\,]\!]$ This lets us derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash k : \mathcal{T}[\![\tau_2]\!] \to \mathcal{T}[\![\tau]\!]}\ \text{T-Var} \quad \dfrac{\dfrac{\text{(1) and induction hypothesis}}{\ldots \vdash \mathcal{H}[\![h]\!]_{\emptyset} : \mathcal{T}[\![\tau_1]\!] \to C[\![\tau_2]\!]_{\emptyset}} \quad \dfrac{}{\ldots, x : \mathcal{T}[\![\tau_1]\!], \ldots \vdash x : \mathcal{T}[\![\tau_1]\!]}\ \text{T-Var}}{\ldots, x : \mathcal{T}[\![\tau_1]\!], \ldots \vdash \mathcal{H}[\![h]\!]_{\emptyset}\ @\ x : \mathcal{T}[\![\tau_2]\!]}\ \text{T-App}}{\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!], x : \mathcal{T}[\![\tau_1]\!], k : \mathcal{T}[\![\tau_2]\!] \to \mathcal{T}[\![\tau]\!] \vdash k\ @\ (\mathcal{H}[\![h]\!]_{\emptyset}\ @\ x) : \mathcal{T}[\![\tau]\!]}\ \text{T-App}}{\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!], x : \mathcal{T}[\![\tau_1]\!] \vdash \lambda k \Rightarrow k\ @\ (\mathcal{H}[\![h]\!]_{\emptyset}\ @\ x) : (\mathcal{T}[\![\tau_2]\!] \to \mathcal{T}[\![\tau]\!]) \to \mathcal{T}[\![\tau]\!]}\ \text{T-Lam}}{\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \lambda x \Rightarrow \lambda k \Rightarrow k\ @\ (\mathcal{H}[\![h]\!]_{\emptyset}\ @\ x) : \mathcal{T}[\![[\![\mathbb{F}]\!]_{\emptyset,\tau}]\!]}\ \text{T-Lam}$$

case Cap-Lift – stack shape $\overline{\tau}, \tau, \tau'$

Similar to the other case, given $\Theta \mid \Gamma \vdash_{\text{cap}} \text{lift } h : [\![\mathbb{F}]\!]_{\overline{\tau},\tau,\tau'}$ we need to show $\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{H}[\![\text{lift } h]\!] : \mathcal{T}[\![[\![\mathbb{F}]\!]_{\overline{\tau},\tau,\tau'}]\!]$.

From the premises, we have $\Theta \mid \Gamma \vdash_{\text{cap}} h : [\![\mathbb{F}]\!]_{\overline{\tau},\tau}$ **(1)**. Further, we can compute:

$\mathcal{T}[\![[\![\mathbb{F}]\!]_{\overline{\tau},\tau,\tau'}]\!] = \mathcal{T}[\![\tau_1]\!] \to C[\![\tau_2]\!]_{\overline{\tau},\tau,\tau'} = \mathcal{T}[\![\tau_1]\!] \to (\mathcal{T}[\![\tau_2]\!] \to C[\![\tau']\!]_{\overline{\tau},\tau}) \to C[\![\tau']\!]_{\overline{\tau},\tau}$

Again, we can derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Ind. hyp., weakening, and (1)}}{\ldots \vdash \mathcal{H}[\![h]\!]_{\overline{\tau},\tau} : \mathcal{T}[\![\tau_1]\!] \to C[\![\tau_2]\!]_{\overline{\tau},\tau}} \quad \dfrac{}{\ldots \vdash x : \mathcal{T}[\![\tau_1]\!]}\,\text{T-Var}}{\ldots \vdash \mathcal{H}[\![h]\!]_{\overline{\tau},\tau} \mathbin{@} x : (\mathcal{T}[\![\tau_2]\!] \to C[\![\tau]\!]_{\overline{\tau}}) \to C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-App} \quad \dfrac{\text{(2)}}{\ldots \vdash \lambda y \Rightarrow k \mathbin{@} y \mathbin{@} k' : \mathcal{T}[\![\tau_2]\!] \to C[\![\tau]\!]_{\overline{\tau}}}}{\ldots, x : \mathcal{T}[\![\tau_1]\!], k : \mathcal{T}[\![\tau_2]\!] \to C[\![\tau']\!]_{\overline{\tau},\tau}, k' : \mathcal{T}[\![\tau']\!] \to C[\![\tau]\!]_{\overline{\tau}} \vdash \ldots : C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-App}}{\ldots, x : \mathcal{T}[\![\tau_1]\!], k : \mathcal{T}[\![\tau_2]\!] \to C[\![\tau']\!]_{\overline{\tau},\tau} \vdash \lambda k' \Rightarrow \ldots : (\mathcal{T}[\![\tau']\!] \to C[\![\tau]\!]_{\overline{\tau}}) \to C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-Lam}}{\ldots, x : \mathcal{T}[\![\tau_1]\!] \vdash \lambda k \Rightarrow \lambda k' \Rightarrow \ldots : (\mathcal{T}[\![\tau_2]\!] \to C[\![\tau']\!]_{\overline{\tau},\tau}) \to C[\![\tau']\!]_{\overline{\tau},\tau}}\,\text{T-Lam}}{\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \lambda x \Rightarrow \lambda k \Rightarrow \lambda k' \Rightarrow \mathcal{H}[\![h]\!]_{\overline{\tau},\tau} \mathbin{@} x \mathbin{@} (\lambda y \Rightarrow k \mathbin{@} y \mathbin{@} k') : \mathcal{T}[\![[\![\mathbb{F}]\!]_{\overline{\tau},\tau,\tau'}]\!]}\,\text{T-Lam}$$

where the typing of the composed continuation **(2)** is given by:

$$\dfrac{\dfrac{\dfrac{}{\ldots \vdash k : \mathcal{T}[\![\tau_2]\!] \to C[\![\tau']\!]_{\overline{\tau},\tau}}\,\text{T-Var} \quad \dfrac{}{\ldots \vdash y : \mathcal{T}[\![\tau_2]\!]}\,\text{T-Var}}{\ldots \vdash k \mathbin{@} y : (\mathcal{T}[\![\tau']\!] \to C[\![\tau]\!]_{\overline{\tau}}) \to C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-App} \quad \dfrac{}{\ldots \vdash k' : \mathcal{T}[\![\tau']\!] \to C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-Var}}{\ldots, y : \mathcal{T}[\![\tau_2]\!] \vdash k \mathbin{@} y \mathbin{@} k' : C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-App}}{\ldots \vdash \lambda y \Rightarrow k \mathbin{@} y \mathbin{@} k' : \mathcal{T}[\![\tau_2]\!] \to C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-Lam}$$

Note that in the derivation, we implicitly expand and contract applications to the meta function $C[\![\cdot]\!]$.

case Cap-Handler

The premises give us $\Theta, k : [\![\text{Resume}_i]\!]_{\overline{\tau}} \mid \Gamma, x : \tau' \vdash s : [\![\tau]\!]_{\overline{\tau}}$ **(1)**, and $\Sigma(\text{Resume}_i) = \tau'' \to \tau$. Using the premises **(1)**, **(3)**, $\mathcal{T}[\![[\![\text{Resume}_i]\!]_{\overline{\tau}} = \mathcal{T}[\![\tau'']\!] \to C[\![\tau]\!]_{\overline{\tau}}$, and the ind. hyp., we obtain: $\mathcal{T}[\![\Theta]\!], k : \mathcal{T}[\![\tau'']\!] \to C[\![\tau]\!]_{\overline{\tau}}, \mathcal{T}[\![\Gamma]\!], x : \mathcal{T}[\![\tau']\!] \vdash \mathcal{S}[\![s]\!]_{\overline{\tau}} : C[\![\tau]\!]_{\overline{\tau}}$ **(3)**.

Now, starting from $\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!] \vdash \lambda x \Rightarrow \lambda k \Rightarrow \mathcal{S}[\![s]\!]_{\overline{\tau}} : \mathcal{T}[\![\tau']\!] \to C[\![\tau'']\!]_{\overline{\tau},\tau}$ we apply T-Lam twice to finally reorder the typing context and apply **(3)**.

Most cases of the typing judgement $\vdash_{\text{exp}}$ are straightforward and we omit them here.

Similar to rule Cap-Lift, rules Ret and Val of judgement $\vdash_{\text{stm}}$ require us to consider two cases: an empty stack shape ($\emptyset$) and a non-empty stack shape ($\overline{\tau}, \tau$). Otherwise, they pose no difficulty. Since handlers delimit the captured continuation, maybe the most interesting case is for rule Handle.

case Handle

From the premises, we obtain $\Theta, c : [\![\mathbb{F}]\!]_{\overline{\tau},\tau} \mid \Gamma \vdash_{\text{stm}} s : [\![\tau]\!]_{\overline{\tau},\tau}$ **(1)** and $\Theta \mid \Gamma \vdash_{\text{cap}} \mathcal{H}[\![h]\!]_{\overline{\tau},\tau} : [\![\mathbb{F}]\!]_{\overline{\tau},\tau}$ **(2)**. Following the syntactic abbreviation, we also assume a derived rule T-Let.

$$\dfrac{\dfrac{\text{(2) and ind. hyp.}}{\ldots \vdash \mathcal{H}[\![h]\!]_{\overline{\tau},\tau} : \mathcal{T}[\![[\![\mathbb{F}]\!]_{\overline{\tau},\tau}]\!]} \quad \dfrac{\dfrac{\text{(1), ind. hyp., and reordering of the context}}{\ldots \vdash \mathcal{S}[\![s]\!]_{\overline{\tau},\tau} : (\mathcal{T}[\![\tau]\!] \to C[\![\tau]\!]_{\overline{\tau}}) \to C[\![\tau]\!]_{\overline{\tau}}} \quad \dfrac{\dfrac{\text{(3)}}{\ldots, x : \mathcal{T}[\![\tau]\!] \vdash \mathcal{S}[\![\text{return } x]\!]_{\overline{\tau}} : C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-Lam}}{\ldots \vdash \lambda x \Rightarrow \mathcal{S}[\![\text{return} x]\!]_{\overline{\tau}} : \mathcal{T}[\![\tau]\!] \to C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-App}}{[\![\Theta]\!], [\![\Gamma]\!], c : \mathcal{T}[\![[\![\mathbb{F}]\!]_{\overline{\tau},\tau}]\!] \vdash \mathcal{S}[\![s]\!]_{\overline{\tau},\tau} \mathbin{@} (\lambda x \Rightarrow \mathcal{S}[\![\text{return } x]\!]_{\overline{\tau}} : C[\![\tau]\!]_{\overline{\tau}}}\,\text{T-App}}{[\![\Theta]\!], [\![\Gamma]\!] \vdash \textbf{let } c = \mathcal{H}[\![h]\!]_{\overline{\tau},\tau} \textbf{ in } \mathcal{S}[\![s]\!]_{\overline{\tau},\tau} \mathbin{@} (\lambda x \Rightarrow \mathcal{S}[\![\text{return } x]\!]_{\overline{\tau}})}\,\text{T-Let}$$

To prove **(3)**, we proceed as with Ret, performing a case distinction on the stack shape $\overline{\tau}$, both times closing in application of T-Var to show $\ldots x : \mathcal{T}[\![\tau]\!] \vdash \mathcal{E}[\![x]\!] : \mathcal{T}[\![\tau]\!]$.

□

Syntax of Terms:
Expressions $e ::= \text{True} \mid x \mid e\ @\ e \mid \lambda x \Rightarrow e \mid \text{letrec}\ f\ =\ e\ \text{in}\ e$

Syntax of Types:
Types $\tau ::= \tau \to \tau \mid \text{Int} \mid \text{Bool} \mid \ldots$

Type Env. $\Gamma ::= \emptyset \mid \Gamma,\ x\ :\ \tau$

Type Rules:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\text{T-Var}] \qquad \frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x \Rightarrow e : \tau_1 \to \tau_2}\ [\text{T-Lam}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ @\ e_2 : \tau_2}\ [\text{T-App}]$$

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \text{letrec}\ f\ =\ e\ \text{in}\ e : \tau}\ [\text{T-Rec}]$$

Fig. 7. The target language – Call-by-value simply typed lambda calculus with letrec.

C.2 Typability of Translated Terms (Staged)

The proof for Theorem 5.3 is structurally similar to the one of Theorem 5.1. Differences are: (a) it uses the typing judgements of 2λ instead of STLC and (b) the translation of lambda abstraction, application and recursive definitions differs.

Proof. We give the case for lambda abstraction, the other cases are similar.

case Lam

From the premise, we obtain $\Theta \mid \Gamma,\ x\ :\ \tau \vdash_{\text{stm}} s\ :\ [\ \tau'\ ]_{\overline{\tau}}$ **(1)**.
Like in the unstaged case, we can derive:

$$\dfrac{\dfrac{\text{Ind. hyp. and } \textbf{(1)}}{\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!], x : \mathcal{T}[\![\tau]\!] \vdash \mathcal{S}[\![s]\!]_{\overline{\tau}} : \overline{C}[\![\tau']\!]_{\overline{\tau}}}\ \textsc{Reify}}{\mathcal{T}[\![\Theta]\!], \mathcal{T}[\![\Gamma]\!], x : \mathcal{T}[\![\tau]\!] \vdash \textsc{Reify}_{\overline{\tau}}\ \mathcal{S}[\![s]\!]_{\overline{\tau}} : \underline{C}[\![\tau']\!]_{\overline{\tau}}}\ \text{T-RLam}$$

Here, we apply the typing rules for residualized abstractions T-RLam followed by Lemma 11.1. We use a Lemma that the translation of a dynamic type $\mathcal{T}[\![\ \tau\ ]\!]$ always results in a residual type in 2λ.

□

Since the staged translation uses Reify and Reflect, we need to adjust our proof accordingly. In particular, we require the following lemma:

Lemma C.1 (Reify / Reflect).

$$\frac{\mathcal{T}[\![\ \Theta\ ]\!],\ \mathcal{T}[\![\ \Gamma\ ]\!] \vdash e : \overline{C}[\![\ \tau\ ]\!]}{\mathcal{T}[\![\ \Theta\ ]\!],\ \mathcal{T}[\![\ \Gamma\ ]\!] \vdash \textsc{Reify}_{\overline{\tau}}\ e : \underline{C}[\![\ \tau\ ]\!]}\ [\textsc{Reify}] \qquad \frac{\mathcal{T}[\![\ \Theta\ ]\!],\ \mathcal{T}[\![\ \Gamma\ ]\!] \vdash e : \underline{C}[\![\ \tau\ ]\!]}{\mathcal{T}[\![\ \Theta\ ]\!],\ \mathcal{T}[\![\ \Gamma\ ]\!] \vdash \textsc{Reflect}_{\overline{\tau}}\ e : \overline{C}[\![\ \tau\ ]\!]}\ [\textsc{Reflect}]$$

Syntax of Terms:

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & \underline{\text{True}} \mid x \\
& & \mid & e \mathbin{\overline{@}} e \mid \overline{\lambda x \Rightarrow e} \\
& & \mid & e \mathbin{\underline{@}} e \mid \underline{\lambda x \Rightarrow e} \\
& & \mid & \underline{\text{letrec}}\ f\ =\ e\ \underline{\text{in}}\ e
\end{array}
$$

Syntax of Types:

$$
\begin{array}{lrcl}
\text{Staged Types} & \sigma & ::= & \sigma \mathbin{\overline{\rightarrow}} \sigma \mid \tau \\
\text{Residual Types} & \tau & ::= & \tau \mathbin{\underline{\rightarrow}} \tau \mid \underline{\text{Int}} \mid \underline{\text{Bool}} \mid \ldots \\
\text{Type Env.} & \Gamma & ::= & \emptyset \mid \Gamma,\ x\ :\ \sigma
\end{array}
$$

Type Rules:

$$\frac{\Gamma(x)\ =\ \sigma}{\Gamma\ \vdash\ x\ :\ \sigma}\ [\text{T-Var}] \qquad \frac{\Gamma,\ x\ :\ \sigma_1\ \vdash\ e\ :\ \sigma_2}{\Gamma\ \vdash\ \overline{\lambda x \Rightarrow e}\ :\ \sigma_1 \mathbin{\overline{\rightarrow}} \sigma_2}\ [\text{T-SLam}] \qquad \frac{\Gamma\ \vdash\ e_1\ :\ \sigma_1 \mathbin{\overline{\rightarrow}} \sigma_2 \quad \Gamma\ \vdash\ e_2\ :\ \sigma_1}{\Gamma\ \vdash\ e_1 \mathbin{\overline{@}} e_2\ :\ \tau_2}\ [\text{T-SApp}]$$

$$\frac{\Gamma,\ x\ :\ \tau_1\ \vdash\ e\ :\ \tau_2}{\Gamma\ \vdash\ \underline{\lambda x \Rightarrow e}\ :\ \tau_1 \mathbin{\underline{\rightarrow}} \tau_2}\ [\text{T-RLam}] \qquad \frac{\Gamma\ \vdash\ e_1\ :\ \tau_1 \mathbin{\underline{\rightarrow}} \tau_2 \quad \Gamma\ \vdash\ e_2\ :\ \tau_1}{\Gamma\ \vdash\ e_1 \mathbin{\underline{@}} e_2\ :\ \tau_2}\ [\text{T-RApp}] \qquad \frac{\Gamma,\ f\ :\ \tau\ \vdash\ e\ :\ \tau}{\Gamma\ \vdash\ \underline{\text{letrec}}\ f\ =\ e\ \underline{\text{in}}\ e\ :\ \tau}\ [\text{T-RRec}]$$

Fig. 8. The target language of the staged translation – 2-level lambda calculus.