Compiling Effect Handlers in Capability-Passing Style

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

Effect handlers encourage programmers to abstract over repeated patterns of complex control flow. As of today, this abstraction comes at a significant price in performance. In this paper, we aim to achieve abstraction without regret for effect handlers.

We present a language for effect handlers in *capability-passing style* (λ_{Cap}) and an implementation of this language as a translation to simply-typed lambda calculus in *iterated continuation-passing style*. A suite of benchmarks indicates that the novel combination of capability-passing style and iterated CPS enables significant speedups over existing languages with effect handlers or control operators. Our implementation technique is general and allows us to generate code in any language that supports first-class functions.

We then identify a subset of programs for which we can further improve the performance and guarantee full elimination of the effect handler abstraction. To formally capture this subset, we refine λ_{Cap} to $\boldsymbol{\lambda}_{\mathrm{Cap}}$ with a more restrictive type system. We present a type-directed translation for $\boldsymbol{\lambda}_{\mathrm{Cap}}$ that inserts staging annotations and prove that no abstractions or applications related to effect handlers occur in the translated program. Using this second translation we observe additional speedups in some of the benchmarks.

CCS Concepts: • **Software and its engineering** → *Source code generation*; *Functional languages*; **Control structures**.

Additional Key Words and Phrases: algebraic effects, control effects, continuations, continuation-passing style, capability-passing style, lexically-scoped effect handlers, compilation

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

1 INTRODUCTION

Effect handlers [Plotkin and Pretnar 2009, 2013] offer high-level control-flow abstractions that are user definable and composable. They have been used successfully to develop libraries for asynchronous programming, libraries for concurrent system programming, programming with coroutines, stream processing, and many more [Bračevac et al. 2018; Dolan et al. 2017, 2015; Hillerström and Lindley 2016; Kammar et al. 2013; Leijen 2017b; Piróg et al. 2018]. 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 [Leijen 2017a; Pretnar et al. 2017]. There are two different aspects to the performance of such abstractions: enabling compile

Philipp Schuster, University of Tübingen, Sand 13, 72076 Tübingen, Germany, philipp.schuster@uni-tuebingen.de; Jonathan Immanuel Brachthäuser, University of Tübingen, Sand 13, 72076 Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de; Klaus Ostermann, University of Tübingen, Sand 13, 72076 Tübingen, Germany, klaus.ostermann@uni-tuebingen.de.

2475-1421/2020/8-ART93
https://doi.org/10.1145/3408975

time optimization [Pretnar et al. 2017; Wu and Schrijvers 2015] and optimizing the language runtime [Leijen 2017a]. 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 [Wright and Felleisen 1994]. In languages with support for effect handlers, the handler implementations are part of this evaluation context. Typically, language runtimes perform 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 and, secondly, capturing a segment of the context delimited by that very handler [Dolan et al. 2014; Hillerström et al. 2017; Kammar et al. 2013; Leijen 2017c]. 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.

In this paper, we present a new approach to efficiently compile effect handlers. We proceed in two steps, reflected in two languages λ_{Cap} and $\lambda\!\!\lambda_{\text{Cap}}$.

The first language, λ_{Cap}, features lexically scoped effect handlers in capability-passing style. Capability-passing style is an alternative to the traditional dynamic lookup for the handler. Instead of searching the implementation on the stack, handler implementations are passed as additional arguments to their use-site. It has been shown [Biernacki et al. 2020; Zhang and Myers 2019] that lexically scoped handlers offer improved reasoning to the programmer. It has also been conjectured [Biernacki et al. 2020] that they may enable more efficient implementations. In this work, we answer this conjecture positively. Using capability passing and iterated continuation passing style (CPS) to implement control yields speedups of up to 150x over existing languages with effect handlers on standard benchmarks from the literature. Intended as a core language, λ_{Cap} makes two kinds of static information explicit. Firstly, it supports effect handlers in *explicit capability-passing style* [Brachthäuser and Schuster 2017; Brachthäuser et al. 2020; Zhang and Myers 2019]. By making the flow of capabilities explicit, an optimizing compiler can specialize programs to known effect handler implementations. 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 [Danvy and Filinski 1990; Schuster and Brachthäuser 2018]. We implement stack shape polymorphism by monomorphization[1], a common technique in performance-oriented compilers [Alexandrescu 2010; Anderson et al. 2016; Stroustrup 1997]. This specializes effectful programs to their stack shape.

The second language, $\lambda\!\!\lambda_{\text{Cap}}$, has the same operational semantics as λ_{Cap} but refines the type system to make capabilities *second class*. This way, the type system restricts the class of programs expressible in $\lambda\!\!\lambda_{\text{Cap}}$ as a sub-language of λ_{Cap} for which we always statically know handler implementations. Yet, it still covers a large class of programs. Statically knowing both the handler for each effect operation and the stack shape, allows us to reduce all abstractions related to effect handlers at compile time (Theorem 5.5). This yields an additional speedup for a total of up to 409x.

To evaluate the performance, in our benchmarks we compare code generated from λ_{Cap} and $\lambda\!\!\lambda_{\text{Cap}}$ with Koka [Leijen 2017c], Multicore OCaml [Dolan et al. 2014], and Chez Scheme [Dybvig 2006]. The benchmarks indicate that our translation offers significant speedups for examples, which

[1]http://mlton.org/Monomorphise

heavily use effects and handlers, and shows competitive performance for examples with only simple uses of effect handlers.

Specifically, we make the following contributions:

- We present the language λ_{Cap} with effect handlers in explicit capability-passing style (Section 3).
- We present a translation from λ_{Cap} to STLC (Section 3.3) that preserves well-typedness (Theorem 5.1). Effect safety follows as a corollary (Corollary 5.2): pure programs do not have unhandled effects at run time.
- Our approach is general as it eliminates the need for a special runtime for effect handlers. We can target any language that supports first class functions.
- We present the language $\lambda\!\!\lambda_{\mathsf{Cap}}$. It treats capabilities as second class and characterizes a sublanguage for which we can guarantee efficient compilation.
- We present a translation for $\lambda\!\!\lambda_{\mathsf{Cap}}$ into a two-level lambda calculus. All abstractions and applications related to effect handlers are statically evaluated and removed from the generated program (Section 4).
- We prove that the translation never fails, always terminates, and that generated programs will never go wrong (Theorem 5.3), which again entails effect safety.
- We prove that translated $\lambda\!\!\lambda_{\mathsf{Cap}}$ programs are free of overhead introduced by the handler abstraction (Theorem 5.5).
- We implemented both, λ_{Cap} and $\lambda\!\!\lambda_{\mathsf{Cap}}$ and performed benchmarks, which suggest that the code we generate is competitive with or faster than Koka, Multicore OCaml, and Chez Scheme code using control effects (Section 5.2).

2 OVERVIEW

In this section, we present an 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 [1990] and written in λ_{Cap}. The effectful function choice uses the two capabilities flip and fail to choose a number between the given argument n and 1. If the 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 [Brachthäuser and Schuster 2017], that is, it explicitly abstracts over the *capabilities* flip and fail. In the body of the function, we use the capabilities (e.g., do flip()) to call an effect operation. Explicitly binding capabilities leads to what Biernacki et al. [2020] refer to as *lexical* effect handlers. Effect handlers (and capability abstractions) are binders and effect operations are resolved lexically.

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

```
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) Source program written in λ_{Cap} in capability-passing style.

```
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
```

```
let handledChoice = λn ⇒
  let flip = λ () ⇒ λk ⇒
    append (k True) (k False) in
  let fail = λ () ⇒ λk₁ ⇒ λk₂ ⇒ k₂ Nil in
  let liftedFlip = λ () ⇒ λk₁ ⇒ λk₂ ⇒
    flip () (λx ⇒ k₁ x k₂) in
    choice liftedFlip fail n
     (λx₁ ⇒ λk₂ ⇒ k₂ (Cons x₁ Nil))
     (λx₂ ⇒ x₂)
```

(b) Code generated from λ_{Cap} in iterated CPS.

```
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)
```

(c) Code generated from $\lambda\!\!\!\lambda_{\text{Cap}}$ with inlined handlers (highlighted in grey).

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

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 [Danvy and Filinski 1990]. 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 call-site 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 [2019], we write the type of the choice function at this call-site as:

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

The function choice 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 $[\ \text{Flip}\ ]_{\text{IntList}}$. 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 (with `letrec`) in iterated continuation-passing style (CPS) [Danvy and Filinski 1990]. Directed by the statically known stack shape, our translation introduces one continuation argument for every delimiting handler. Figure 1b shows the result of specializing choice to the stack shape IntList, IntList at its call-site. The generated code uses two continuations corresponding to the two delimiters for Flip and Fail. At its call-site we supply five arguments to choice: two capabilities, the argument n, and two continuations corresponding to the two delimiters. The first continuation represents the context around choice at its call-site. It is itself in CPS and takes a continuation. The second continuation is the empty continuation. Capability flip has been translated at stack shape IntList and so abstracts over only one continuation. We now see that lift has operational meaning as it adjusts flip to be compatible to a context with two continuations by composing them.

If we were using the same choice function at a different call-site, within for example three enclosing handlers, it would be typed at a different stack shape and consequently specialized differently:

```
let choice = λflip ⇒ λfail ⇒
  letrec loop = λn ⇒ λk₁ ⇒ λk₂ ⇒ λk₃ ⇒
    if (n < 1) then fail () k₁ k₂ k₃
    else flip () (λx ⇒ λk₄ ⇒ λk₅ ⇒ if x then k₁ n k₄ k₅ else loop (n − 1) k₁ k₄ k₅ ) k₂ k₃
  in loop
```

We abstract over one more continuation and apply functions to one more continuation (highlighted in grey). Through monomorphization we have created a second specialized version of the same function. Operationally, only the number of elements in the stack shape, i.e. the number of continuations, matters, not their order. In a typed setting, though, we have to specialize to the types contained in the stack shape. While this translation specializes choice to different stack shapes, it still abstracts over capabilities flip and fail.

Compilation of $\lambda\!\!\lambda_{\mathsf{Cap}}$. Our running example does not treat capabilities as first class [Osvald et al. 2016] and so can be typed under the more restrictive rules of $\lambda\!\!\lambda_{\mathsf{Cap}}$, which enforce a second-class usage of capabilities. Consequently, we can use our second translation, which works for $\lambda\!\!\lambda_{\mathsf{Cap}}$ only, to also specialize the code to the concrete handler implementations. In this translation we distinguish between static and dynamic abstractions 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 1c shows the final code we generate for this example. We chose the name choiceFlipFail to reflect the specialization to these handler implementations. 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 [Leijen 2017c], Eff [Plotkin and Pretnar 2013], Frank [Lindley et al. 2017], Multicore OCaml [Dolan et al. 2014], or Helium [Biernacki et al. 2020]. Instead, the implementations of the effect operations have been inlined into the body of choice. Correspondingly, the call-site in handledChoice does not provide capabilities anymore, it only delimits the control effects. There is no search for a delimiter on the stack, either. We directly invoke the corresponding continuation. Finally, lifting of the capability is performed during translation.

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

Syntax of Terms:

Expressions

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

Statements

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

Capabilities

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

Syntax of Types:

Types

τ	$::=$	$\mathsf{Int} \mid \mathsf{Bool} \mid \ldots$	base types
	$\mid$	$\tau \rightarrow [\, \tau \,]_{\overline{\tau}}$	effectful function type
	$\mid$	$[\, \mathbb{F} \,]_{\overline{\tau}} \rightarrow \tau$	capability function type

Operation Names

$\mathbb{F} \ ::= \ \mathsf{Flip} \mid \mathsf{Fail} \mid \mathsf{Emit} \mid \mathsf{Resume}_i \mid \ldots$

Operation Signatures

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

Type Environment

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

Capability Environment

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

Stack Shape

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

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

adjust them to the stack shape. Using this information, we specialize functions written in λ_{Cap} to work with the correct number of continuations. For a refined sub-language $\mathbb{\lambda}_{\mathsf{Cap}}$, we guarantee that capabilities are always inlined. We specialize functions to their context and remove the cost associated with handler abstractions. In the following sections we will formally develop these ideas.

3 CAPABILITY PASSING

In this section, we formally introduce λ_{Cap} – a language with effects, handlers, and capabilities. The presentation follows the calculus by Zhang and Myers [2019] with some notable differences discussed in Section 6.1.

3.1 Syntax of Terms

Figure 2 defines the syntax of λ_{Cap}. Like other presentations of languages with effect handlers [Hillerström et al. 2017; Kammar and Pretnar 2017; Pretnar 2015], our language is based on a fine-grain call-by-value lambda calculus [Levy et al. 2003]. 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" [Reynolds 1972]. Other than most effect languages, which do not represent capabilities explicitly, we also syntactically distinguish between expressions and capabilities.

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. $\mathsf{fix}\, f\ ((x : \tau) \Rightarrow s)$). 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 $\text{append}(e,\ e)$) 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. The handler also installs a delimiter for the continuation that is captured when c is used.

Capabilities. We 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. This stratification into expressions and capabilities is not strictly necessary in λ_{Cap}, but it will become important in $\mathbb{\lambda}_{\text{Cap}}$. To facilitate comparison we use the same syntax of terms for both languages. 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}

Types include base types (e.g., Int and Bool), effectful function types $\tau \to [\ \tau'\ ]_{\overline{\tau}}$, and capability abstractions $[\ \mathbb{F}\ ]_{\overline{\tau}} \to \tau$. 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 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 [Leijen 2017c] or Links [Hillerström and Lindley 2016]. 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 $\mathsf{Int}, \mathsf{String}$ describes a context with an outer handler at type Int and an inner handler at type String. Capability abstractions take a capability parameter. 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 $\text{handle} \ldots \text{in} \ldots$ 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 types τ and a capability environment Θ that associates capability variables c with operation names $\mathbb{F}$ and stack shapes $\overline{\tau}$. This, again, is not necessary in λ_{Cap} but will be in $\mathbb{\lambda}_{\text{Cap}}$.

3.2.1 *Typing Rules.*

The typing rules in Figure 3 are defined by three mutually recursive typing judgements – one for each syntactic category. The judgement form $\Theta \mid \Gamma\ \vdash_{\text{stm}}\ s\ :\ [\ \tau\ ]_{\overline{\tau}}$ assigns a pair of a 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.

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Typing rules for λ_{Cap}.

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. In rule Ret, the resulting statement is compatible with any stack shape $\overline{\tau}$.

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 capability 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 operation name $\mathbb{F}$ in stack shape $\overline{\tau}$.

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 [Convent et al. 2020], to *lift* in Helium [Biernacki et al. 2019], and to *inject* in Koka [Leijen 2018]. 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 $\tau'' \to [\ \tau\ ]_{\overline{\tau}}$. We model resumptions as effect operations. The body s of the handler is evaluated in stack shape $\overline{\tau}$, that is, outside of the delimiter that introduced it.

3.3 Translation of λ_{Cap} to STLC

In this subsection, we describe the semantics of λ_{Cap} in terms of a translation to simply-typed lambda calculus [Barendregt 1992], extended with a standard **letrec** operator to express fix. We translate λ_{Cap} into iterated CPS where capabilities are still present at runtime. In the translation of $\boldsymbol{\lambda}_{\text{Cap}}$ (Section 4.2), we use a two-level lambda calculus as the target, marking some abstractions as static and others as dynamic. This allows us to prevent administrative redexes and, more importantly, to eliminate all abstractions related to handlers and capabilities.

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 [2018] to the setting of effect handlers. At its heart, our translation is thus an iterated CPS translation [Danvy and Filinski 1990] but building on the control operator *shift*$_0$ [Danvy and Filinski 1989; Materzok and Biernacki 2012] rather than *shift*, because it more closely fits effect handlers [Forster et al. 2017; Hillerström et al. 2017; Kammar et al. 2013]. In Theorem 5.1, we show that our translation takes well-typed λ_{Cap} programs to well-typed STLC programs.

3.3.1 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 [Nielson and Nielson 1996]. We sometimes use **let** bindings in the target language assuming the standard shorthand: **let** x = e **in** e′ $\doteq$ (λx ⇒ e′) @ e.

3.3.2 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 $\mathcal{C}[\![\ \tau'\ ]\!]_{\overline{\tau}}$. 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 $\mathcal{C}[\![\ \tau\ ]\!]_{\overline{\tau}}$ 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:

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

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

Translation of Types:

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

Translation of Expressions:

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

Translation of Statements:

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

Translation of Capabilities:

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

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

3.3.3 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.

3.3.4 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 $\mathcal{C}[\![\ \tau\ ]\!]_{\overline{\tau}}$. In the case of a pure statement without effects, the stack shape is empty and we do not perform a CPS translation. Returning translates to just the returned expression and, to preserve sharing of results, sequencing translates to a let binding. 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 ($\mathsf{do}\ h(e)$) to a function application. The translation of $\mathsf{handle}\ c\ =\ h\ \mathsf{in}\ s$ 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.

3.3.5 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:

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

Assuming an answer type of Int, we obtain:

$$\begin{array}{l}\textbf{let}\ c\ =\ \mathcal{H}[\![\ h\ ]\!]_{\mathsf{Int}}\ \textbf{in}\\ \quad \underbrace{\mathcal{S}[\![\ \mathsf{val}\ x \leftarrow \mathsf{do}\ c(\mathsf{True});\ \mathsf{return}\ e\ ]\!]_{\mathsf{Int}}}_{\lambda k \Rightarrow \mathcal{S}[\![\ \underbrace{\mathsf{do}\ c(\mathsf{True})}_{c\ @\ \mathsf{True}}\ ]\!]_{\mathsf{Int}}\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\ \underbrace{\mathsf{return}\ e}_{\lambda k' \Rightarrow k'\ @\ \mathcal{E}[\![\ e\ ]\!]}\ ]\!]_{\mathsf{Int}}\ @\ k)}\ @\ (\lambda x \Rightarrow \mathcal{S}[\![\ \mathsf{return}\ x\ ]\!]_{\emptyset})\end{array}$$

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

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

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

4 ZERO COST EFFECT HANDLERS

We now refine λ_{Cap} to a sub-language $\boldsymbol{\lambda}_{\mathsf{Cap}}$. On this sub-language we are able to fully eliminate the overhead introduced by handler abstractions. We present a second translation, this time to 2-level lambda calculus. This allows us to prove that all abstractions and applications related to effect handlers will be statically reduced at compile time.

Syntax of Types:

Dynamic Types

$$
\begin{array}{lcll}
\tau & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \ldots & \text{base types} \\
 & \mid & \tau \rightarrow [\,\tau\,]_{\overline{\tau}} & \text{effectful function type}
\end{array}
$$

Static Types

$$
\begin{array}{lcll}
\sigma & ::= & [\,\mathbb{F}\,]_{\overline{\tau}} \rightarrow \sigma & \text{capability function type} \\
 & \mid & \tau & \text{dynamic type}
\end{array}
$$

Operation Names

$$
\mathbb{F} \;::=\; \mathsf{Flip} \mid \mathsf{Fail} \mid \mathsf{Emit} \mid \mathsf{Resume}_i \mid \ldots
$$

Operation Signatures

$$\Sigma \;::=\; \emptyset \mid \Sigma,\ \mathbb{F} : \tau \rightarrow \tau'$$

Type Environment

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

Capability Environment

$$\Theta \;::=\; \emptyset \mid \Theta,\ c : [\,\mathbb{F}\,]_{\overline{\tau}}$$

Stack Shape

$$\overline{\tau} \;::=\; \emptyset \mid \overline{\tau},\ \tau$$

Typing Rules:

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

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

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

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

Fig. 5. Syntax of types and expression typing rules for $\mathbb{\lambda}_{\text{Cap}}$ – syntax of terms and typing rules for statements and capabilities are the same as for λ_{Cap}.

4.1 Syntax of $\mathbb{\lambda}_{\text{Cap}}$

Figure 5 lists the syntax of types of $\mathbb{\lambda}_{\text{Cap}}$. The syntax of terms is exactly the same as the one for λ_{Cap}. In the syntax of types we now distinguish between *dynamic types* and *static types*, similarly to the syntactic separation of expressions and capabilities (this difference is highlighted in grey). Static types are sequences of capability parameters ending in a dynamic type, which ensures that all capability arguments come before any other arguments. This is the only difference to λ_{Cap} in Figure 2. Later in this section, we will see that terms of static types will be eliminated (due to inlining and specialization) during translation while terms of dynamic types will appear in the generated program.

Changes to the typing rules for $\mathbb{\lambda}_{\text{Cap}}$ (Figure 5) compared to λ_{Cap} (Figure 3) are also highlighted in grey. Importantly, while the judgement form $\Theta \mid \Gamma \vdash_{\text{exp}} e : \sigma$ may assign a static type σ to expressions, the typing rules for statements and capabilities remain unchanged. Their premises still 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.

We treat capabilities as second class [Osvald et al. 2016]. That is, they cannot be returned from a function. This becomes evident in the typing rules. In the rule Ret (Figure 3):

$$\frac{\Theta \mid \Gamma \vdash_{\text{exp}} e : \tau}{\Theta \mid \Gamma \vdash_{\text{stm}} \mathsf{return}\ e : [\,\tau\,]_{\overline{\tau}}}\ [\textsc{Ret}]$$

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. Similarly, argument expressions in rule App are required to be of a dynamic type τ', which means that we cannot abstract over capability abstractions.

We model resumptions as capabilities, which makes them second class. This is in order to guarantee full elimination of the handler abstraction at compile time. Since capabilities will be inlined, so will resumptions.

4.2 Translation of $\lambdabar_{\mathsf{Cap}}$ to 2-level Lambda Calculus

The programs generated by the translation of λ_{Cap} in Figure 4 abstract over handler implementations and pass them along at runtime. In the translation of $\lambdabar_{\mathsf{Cap}}$, we avoid this passing of capabilities at run time and statically specialize functions to the capabilities that they use. Maybe more importantly, we also specialize the inlined capabilities to the *context* they are used in. This enables optimizations across effect calls.

Figure 6 presents the refined translation from $\lambdabar_{\mathsf{Cap}}$ to 2-level lambda calculus [Jones et al. 1993; Nielson and Nielson 1996; Taha and Sheard 2000]. The translation is the same as the one in Figure 4 except for annotations to distinguish static from dynamic program fragments. The annotations are automatically inserted as part of the definition of our translation. In Theorem 5.3 we prove "stage-time correctness", i.e., that we never confuse static and dynamic functions. This is only possible because the type system of $\lambdabar_{\mathsf{Cap}}$ restricted the use of capabilities, making them second class.

We want to guarantee that certain redexes never occur in the generated program. In particular, there are two classes of redexes that we want to avoid. Firstly, we avoid generating administrative beta redexes in our CPS translation. This standard use multi-level lambda calculi in the translation of control operators has been introduced by Danvy and Filinski [1992]. We build on a variant, which is generalized to the setting of iterated CPS [Schuster and Brachthäuser 2018]. Even though not listed in Figure 4, in our benchmarks we also do this for the unrestricted language in Figure 2. Secondly, we avoid generating redexes associated with the effect handler abstraction. The significant contribution of this paper is that handling effects, calling effect operations, and lifting capabilities does not introduce *any* redexes in the generated program.

4.2.1 *2-level Lambda Calculus.*

The general idea of multi-level lambda calculi [Nielson and Nielson 1996] 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 adopt the terminology of Taha and Sheard [1997] and refer to the annotations as *staging annotations*. 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 $\underline{\mathsf{Int}} \mathrel{\underline{\to}} \underline{\mathsf{Int}}$ 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 underline. 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 $(\overline{\lambda x \Rightarrow}\, x)\ \overline{@}\ \underline{5}$ will statically evaluate to the term $\underline{5}$. We use $\underline{C}[\![\, \tau \,]\!]_{\underline{\tau}}$ to describe the type of residual effectful computations. The whole computation type $C[\![\, \tau \,]\!]_{\underline{\tau}}$ as defined in Figure 4 is residualized. The type $\overline{C}[\![\, \tau \,]\!]_{\underline{\tau}}$ represents static computations. Importantly, while the answer types are residual (e.g., $\underline{\tau}$) the structure of the computation is static.

Translation of Types:

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

Translation of Expressions:

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

Translation of Statements:

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

Translation of Capabilities:

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

Fig. 6. Translation of λ_{Cap} to 2-level lambda calculus.

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

$$
\begin{array}{lcl}
\textsc{Reify}_{\,\overline{\tau}} & : & \overline{C}\llbracket\, \tau \,\rrbracket_{\,\overline{\tau}} \to \underline{C}\llbracket\, \tau \,\rrbracket_{\overline{\tau}} \\
\textsc{Reify}_{\emptyset}\ s & \doteq & s \\
\textsc{Reify}_{\,\overline{\tau},\tau}\ s & \doteq & \underline{\lambda k \Rightarrow}\ \textsc{Reify}_{\overline{\tau}}\ (s\ \overline{@}\ (\overline{\lambda x \Rightarrow}\ \textsc{Reflect}_{\overline{\tau}}\ (k\ \underline{@}\ x)))
\end{array}
$$

$$
\begin{array}{lcl}
\textsc{Reflect}_{\,\overline{\tau}} & : & \underline{C}\llbracket\, \tau \,\rrbracket_{\,\overline{\tau}} \to \overline{C}\llbracket\, \tau \,\rrbracket_{\overline{\tau}} \\
\textsc{Reflect}_{\emptyset}\ s & \doteq & s \\
\textsc{Reflect}_{\,\overline{\tau},\tau}\ s & \doteq & \overline{\lambda k \Rightarrow}\ \textsc{Reflect}_{\overline{\tau}}\ (s\ \underline{@}\ (\underline{\lambda x \Rightarrow}\ \textsc{Reify}_{\overline{\tau}}\ (k\ \overline{@}\ x)))
\end{array}
$$

The meta function Reify converts a static computation of type $\overline{C}\llbracket\, \tau \,\rrbracket_{\overline{\tau}}$ to a residual computation of type $\underline{C}\llbracket\, \tau \,\rrbracket_{\overline{\tau}}$. 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 $\underline{C}\llbracket\, \tau \,\rrbracket_{\overline{\tau}}$ to a static computation of type $\overline{C}\llbracket\, \tau \,\rrbracket_{\overline{\tau}}$. 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.

4.2.3 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.

4.2.4 Statements. We translate statements typed against $[\ \tau\ ]_{\overline{\tau}}$ to *static* computations of type $\overline{C}\llbracket\, \tau \,\rrbracket_{\overline{\tau}}$. 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. 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.

4.2.5 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 be compatible 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 Section 3.3, we obtain

$$
\begin{array}{l}
\overline{\mathsf{let}}\ c\ =\ \mathcal{H}\llbracket\, h \,\rrbracket_{\mathsf{Int}}\ \overline{\mathsf{in}} \\
\overline{\lambda k \Rightarrow}\ c\ \overline{@}\ \underline{\mathsf{True}}\ \overline{@}\ (\overline{\lambda x \Rightarrow}\ (\overline{\lambda k' \Rightarrow}\ k'\ \overline{@}\ \mathcal{E}\llbracket\, e \,\rrbracket)\ \overline{@}\ k)\ \overline{@}\ (\overline{\lambda x \Rightarrow}\ x)
\end{array}
$$

which reduces statically to:

$$
\mathcal{H}\llbracket\, h \,\rrbracket_{\mathsf{Int}}\overline{@}\ \underline{\mathsf{True}}\ \overline{@}\ (\overline{\lambda x \Rightarrow}\ \mathcal{E}\llbracket\, e \,\rrbracket)
$$

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:

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

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.

4.3 Abstracting over Handlers

Other than in λ_{Cap}, capabilities in $\bar{\lambda}_{\mathsf{Cap}}$ are *second class* [Osvald et al. 2016]. This allows us to prove that they never appear in translated programs, but prevents us from writing certain kinds of programs in $\bar{\lambda}_{\mathsf{Cap}}$. In particular, we cannot abstract over handlers as handler functions, a common idiom in Koka for example. Consider the following example written in λ_{Cap} following this idiom:

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

We define a handler function that handles the Fail effect by discarding the continuation and answering with the empty list. This handler is useful and such a definition might be part of the standard library. However, this example is ruled out by the more restrictive type system of $\bar{\lambda}_{\mathsf{Cap}}$. Being a parameter, prog has a dynamic type, but capability application prog[fail] demands that prog has a static type. We thus cannot define handlers as handler functions in $\bar{\lambda}_{\mathsf{Cap}}$.

We can, however, still define and reuse handlers in multiple places. Consider the following example using a hypothetical language construct defhandler:

```
defhandler handleFailList = Fail((), k) ⇒ Nil in
...
handle fail = handleFailList in ...
...
```

This language construct is macro-expressible in $\bar{\lambda}_{\mathsf{Cap}}$ as

$$\mathsf{defhandler}\ \mathsf{c} = \mathsf{h}\ \mathsf{in}\ \mathsf{e} \ \doteq\ ([\mathsf{c}] \Rightarrow \mathsf{e})\ \mathsf{h}$$

Because capability abstractions are reduced statically, the newly defined handler will be inlined at all of its call sites, maintaining the guarantee that functions are specialized to the effect handlers.

5 EVALUATION

We implemented[3] λ_{Cap} and $\lambda\!\!\lambda_{\mathsf{Cap}}$ as *shallow embeddings* into the dependently typed programming language Idris [Brady 2013]. We use typed HOAS [Pfenning and Elliot 1988] 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. For example, the type $\underline{\mathsf{Int}} \mathbin{\underline{\rightarrow}} \underline{\mathsf{Int}}$ would correspond to the Idris type $\mathtt{Exp}\ (\mathsf{Int} \rightarrow \mathsf{Int})$ and the type $\underline{\mathsf{Int}} \mathbin{\overline{\rightarrow}} \underline{\mathsf{Int}}$ would correspond to the Idris type $\mathtt{Exp}\ \mathsf{Int} \rightarrow \mathtt{Exp}\ \mathsf{Int}$. 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. For example the λ_{Cap} type $\mathsf{Int} \rightarrow [\ \mathsf{Bool}\ ]_{\mathsf{Int},\mathsf{String}}$ would correspond to the Idris type $\mathtt{Exp}\ (\mathsf{Int} \rightarrow \mathtt{Eff}\ [\mathsf{String}, \mathsf{Int}]\ \mathsf{Bool})$. Our translation follows the inductive structure of this type-level list.

5.1 Theoretical Results

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

Theorem 5.1 (Typability of translated terms – unstaged).

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

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

Importantly, we obtain effect safety as corollary.

Corollary 5.2 (Effect safety). *Given a closed statement 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 6).

Theorem 5.3 (Typability of translated terms – staged).

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

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

Our translation thus 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*: the 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.*

[3]We submitted our implementation, generated code, benchmarking code and measurements as supplementary material.

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 a residual type $\underline{\tau'}$. Soundness of the 2-level lambda calculus guarantees that stage-time reduction will not get stuck. Our translation does not introduce any static **letrec**, which guarantees termination. □

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 in $\lambda\!\!\!\lambda_{\text{Cap}}$ allows us to guarantee that abstracting over effect operations with handlers does not incur any runtime overhead.

Theorem 5.5 (Full elimination). *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 6. 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 assess the performance of the code generated from λ_{Cap} and $\lambda\!\!\!\lambda_{\text{Cap}}$ and compare it to existing languages with effect handlers and control effects. The benchmarked programs (Triple, Queens, Count, and Generator) can be expressed in both λ_{Cap} and $\lambda\!\!\!\lambda_{\text{Cap}}$. All except for Generator are taken from the literature.

The results are shown in Figure 7. All benchmarks were executed on a 2.60GHz Intel(R) Core(TM) i7 with 11GB of RAM. We compare our implementations with Koka (0.9.0) [Leijen 2017c], Multicore OCaml (4.06.1) [Dolan et al. 2014], and an implementation of delimited control operators in Chez Scheme (9.5.3) [Dybvig et al. 2007]. 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 using the translations of λ_{Cap} (Figure 4) and $\lambda\!\!\!\lambda_{\text{Cap}}$ (Figure 6). In our implementation of both translations, we additionally apply standard techniques [Danvy and Filinski 1992; Schuster and Brachthäuser 2018] to avoid generating administrative eta redexes, although these are not shown in Figure 4. We report the mean and standard deviation of the runtime of the programs under consideration.

We report numbers for four example programs. The Triple program is inspired by the example in Danvy and Filinski [1990]. It uses the running example 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 [2018]. The Count benchmark appears in Kammar et al. [2013], Kiselyov and Ishii [2015], and Wu and Schrijvers [2015] and counts down

Benchmark	Time in ms (Standard Deviation)			
	Baseline	λ_{Cap}	$\mathbb{\lambda}_{\text{Cap}}$	**Native**
Koka				
Triple	2504.1 ±19.5	66.2 ±2.2	23.9 ±0.6	6.2 ±0.2
Queens (18)	403.4 ±9.3	170.8 ±1.7	171.4 ±1.2	161.9 ±4.1
Count (2K)	56.0 ±1.8	0.4 ±0.0	0.2 ±0.0	0.0 ±0.0
Generator (1K)	43.9 ±1.8	0.4 ±0.0	0.1 ±0.0	0.0 ±0.0
Chez Scheme				
Triple	68.6 ±1.1	3.7 ±0.1	3.7 ±0.1	1.8 ±0.0
Queens (18)	93.7 ±3.5	89.6 ±0.6	88.1 ±1.0	89.5 ±1.2
Count (1M)	445.2 ±27.2	10.5 ±0.6	10.5 ±0.8	1.9 ±0.0
Generator (1M)	664.2 ±14.6	17.6 ±0.5	17.7 ±0.5	2.1 ±0.0
Multicore OCaml				
Triple	25.0 ±2.4	4.5 ±0.1	2.4 ±0.1	2.0 ±0.1
Queens (18)	57.9 ±2.2	33.1 ±0.7	33.7 ±0.6	34.8 ±2.7
Count (1M)	72.5 ±0.9	19.4 ±0.5	7.5 ±0.2	2.8 ±0.0
Generator (1M)	93.9 ±1.3	18.3 ±0.5	10.3 ±0.3	3.9 ±0.1
Primes (1K)	32.2 ±0.6	29.0 ±0.6	22.8 ±0.4	N/A
Chameneos	26.7 ±0.6	32.7 ±1.0	28.7 ±0.9	N/A

Fig. 7. Comparing the performance of λ_{Cap} and $\mathbb{\lambda}_{\text{Cap}}$ with Koka, Multicore OCaml, and Chez Scheme.

recursively using a single state effect. The Generator benchmark uses an effect operation to yield numbers which are summed up by a calling function using a state effect.

We now discuss the setup and observations specific to each of the baselines we compare against.

Comparison with Koka. Koka compiles to JavaScript and uses a standard library of builtin functions and data types also compiled to JavaScript. In our comparison with Koka, we do not generate Koka but JavaScript code in CPS and use the same compiled standard library. Benchmarks were executed using the JavaScript library `benchmark.js`[4] on `Node.js`[5] version 12.11.1. For the Count and Generator benchmarks, 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 [Dolan et al. 2014] is a fork of the OCaml compiler [Leroy et al. 2017] 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 `ocamlopt` compiler (4.06.1). Each program is compiled to a standalone executable, and we measure the running time with the `bench` program[6]. In our comparison with Multicore OCaml, we benchmarked two additional examples from an online repository of Multicore OCaml examples[7]: Chameneos and Primes. These two benchmarks exercise the use-case that Multicore OCaml was designed for, that is, resuming continuations only once. Our translation always supports resuming continuations multiple times, but still offers competitive performance. The two additional examples use native

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

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 Monadic Delimited Control 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` [Danvy and Filinski 1989]. We use the library described by [Dybvig et al. 2007] and compile the example programs as well as our generated code with the Chez Scheme compiler [Dybvig 2006]. In all four benchmarks we do not observe any speedup of the code generated from the translation of λ_{Cap} over the code generated from the translation of $\lambda\!\!\lambda_{\text{Cap}}$ where we eliminate redexes during translation. We have investigated Chez Scheme's intermediate representation and confirmed that, after optimization, the code is indeed the same for λ_{Cap} and $\lambda\!\!\lambda_{\text{Cap}}$, except that sometimes a subexpression is let bound. Does this make the restriction of $\lambda\!\!\lambda_{\text{Cap}}$ and its translation in Figure 6 unnecessary? No, on the contrary: the type system of $\lambda\!\!\lambda_{\text{Cap}}$ is an important conceptual tool that guided us to a well-performing implementation where optimal compilation is guaranteed. The fact that Chez Scheme can optimize the program similar to our improved translation (Figure 6) can be seen as additional practical evidence for Theorem 5.5.

Benchmark results. The benchmark results are generally encouraging. They indicate that the code we generate for $\lambda\!\!\lambda_{\text{Cap}}$ (and λ_{Cap}) is significantly faster than the languages we compare against. For the Triple benchmark, we can observe speedups of 105x (λ_{Cap} 38x) compared to Koka, 11x (λ_{Cap} 6x) compared to Multicore OCaml, and 19x for both implementations compared to Chez Scheme. For the Count benchmark we observe speedups of 297x (λ_{Cap} 150x) compared to Koka, 10x (λ_{Cap} 4x) compared to Multicore OCaml, and 42x (λ_{Cap} 43x) compared to Chez Scheme. For the Generator benchmark we observe speedups of 409x (λ_{Cap} 118x) compared to Koka, 9x (λ_{Cap} 5x) compared to Multicore OCaml, and 38x (λ_{Cap} 38x) compared to Chez Scheme. All three benchmarks extensively use control effects and yield optimization opportunities across effect operations for us to exploit. In the other benchmarks, we observe some speedups as well. Interestingly, in the Queens benchmark we do not observe any speedup between our unstaged translation and our staged translation. It uses one effect operation in a single place and handles it as a loop, which our staged translation immediately residualizes.

Comparison with hand-written code using native effects. As another point of reference, we have implemented the benchmark examples and manually restructured the programs to avoid effect handlers and use native effects instead. For example, we used native mutable state instead of effect handlers. We were careful to keep the same number of library calls (for example append on lists). The results are listed in column "Native" in Figure 7. In the benchmarks for Koka, where we compare with the generated code in JavaScript, the time for the native benchmarks is below 0.05 and thus displayed as 0.0. The results indicate that there is still an order of magnitude difference between the code we generate as the translation of $\lambda\!\!\lambda_{\text{Cap}}$ and the hand-written code using native effects.

6 RELATED WORK

We combine capability passing with an implementation of control effects by iterated CPS transformation. This combination allows us to exploit static information and enables compile-time optimizations of effect handlers. Applying some restrictions in $\lambda\!\!\lambda_{\text{Cap}}$, we are able to guarantee that all overhead introduced by the effect handler abstraction is eliminated. In this section, we relate our approach to existing work on capability passing, on compilation of effect handlers via CPS transformation, on optimization of programs which use effect handlers, and on monad transformers.

6.1 Capability Passing

Capability passing for effect handlers is not a new idea. Brachthäuser et al. [2017;2018;2020] establish *capability-passing style* as an implementation technique for effect handlers. They do not present a formal calculus but use capability passing in their library embeddings of effect handlers. To capture the continuation, they use a monadic implementation of multi-prompt delimited continuations [Dybvig et al. 2007]. Their capabilities are pairs of the handler implementation and a prompt. The latter is necessary to enable capturing the part of the stack up to the corresponding delimiter. Passing capabilities explicitly facilitates optimizations by the JVM. By using iterated CPS, we go further and enable optimizations across effect calls.

Zhang and Myers [2019] present a language $\lambda_{\Downarrow\Uparrow}$ that employs capability passing (handler passing) for *abstraction safety*. They demonstrate modular reasoning about effect-polymorphic higher-order functions. Our calculus λ_{Cap} is modeled after $\lambda_{\Downarrow\Uparrow}$. 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.

Biernacki et al. [2020] build on the work by Brachthäuser and Schuster [2017] and Zhang and Myers [2019] and present a language with lexically scoped effects. They argue that lexically scoped effects improve reasoning. Explicitly binding effects also allows to refer to one particular effect instance in the presence of multiple copies of the same effect. Their operational semantics does not employ capability passing as they look up handler implementations based on a label when an effect operation is called. Another difference is that they use an implementation of multi-prompt delimited control to get access to the current continuation, while we translate to iterated CPS.

Kammar et al. [2013] 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 some form of 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. However, they rely on GHC to optimize the abstractions they introduce. They do not explicitly state their assumptions for efficient code generation while, with $\mathbb{\lambda}_{\mathsf{Cap}}$, we make such assumptions explicit.

6.2 Implementing Control-Effects by CPS Translation

Hillerström et al. [2017] present an implementation technique for effect handlers by CPS transformation. They also use a two-level lambda calculus to remove administrative redexes of the CPS translation. An important difference is that their source language has dynamically bound handler implementations, while we support lexically 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. Furthermore, in the translation presented by Hillerström et al., functions are not specialized to their calling context. In consequence, continuation capture across function boundaries still incurs significant runtime overhead, while we fully remove the overhead of the handler abstraction. They report their curried CPS translation to be a composition of an implementation of effect handlers [Forster et al. 2017] and an implementation of delimited continuations in terms of a CPS transformation [Materzok and Biernacki 2012]. Similarly, our type system and translation into iterated CPS is also close to the one by Materzok and Biernacki [2012]. However, we simplify the type system by not supporting answer-type modification, which makes our stack shapes lists rather than trees.

Leijen [2017c] 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 continuation arguments.

We follow Schuster and Brachthäuser [2018] 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.

6.3 Optimization of Effect Handlers

Pretnar et al. [2017] 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" [Saleh et al. 2018]. 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 [2015] 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, but do not state the conditions for when this may or may not happen. To get access to the current continuation, they use nested layers of the codensity monad which is operationally the same as the continuation monad. This is a similarity to our translation to nested layers of CPS.

6.4 Monad Transformers

Our lift h construct is remindful of lifting monad transformers [Liang et al. 1995]. Monad transformers have been proposed as a modular way for writing interpreters for languages with different effects. Today, monad transformers are used in Haskell as a library for effectful programming. Effect handlers, similarly, can be used to modularly define interpreters with different effects and can be embedded into languages as libraries for effectful programming. Monad transformers overload monadic sequencing and returning specially for each effect and, moreover, the definition of lifting depends on the lifted effects. In contrast, our translation of sequencing and lifting is always the same (except in the pure case), regardless of the concrete effects. Practical uses of monad transformers in Haskell heavily rely on inlining and specialization to exhibit good performance. However, the conditions under which this specialization does or does not happen are not clearly stated. We precisely specify the conditions under which we guarantee full elimination of effect handlers.

7 CONCLUSION AND FUTURE WORK

We have presented λ_{Cap}, a language with effect handlers in explicit capability-passing style. We then presented a second language $\mathbb{\lambda}_{\text{Cap}}$, whose type system restricts programs to make it possible to always statically know handler implementations. We have given a translation of λ_{Cap} to STLC that generates fast code. The translation of $\mathbb{\lambda}_{\text{Cap}}$ exploits this static knowledge to eliminate all overhead introduced by abstracting over effect operations. The crucial ingredients are capability passing and iterated CPS.

Since λ_{Cap} and $\mathbb{\lambda}_{\text{Cap}}$ expose 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} and how to detect parts in the sub-language $\mathbb{\lambda}_{\text{Cap}}$.

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.

We see potential for improvement in the future. It is common practice to compile to CPS [Kennedy 2007], an explicit representation of join points [Maurer et al. 2017], or both [Cong et al. 2019]. 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 [Kennedy 2007] generalizing from two continuations to an arbitrary number.

Implementing programming languages involves a series of tradeoffs, usually on spectrum between dynamic and static. This is no different for the implementation of effect handlers. By using capability passing, iterated CPS, and monomorphizing effect-polymorphic function, we have explored the static end of this spectrum in detail and offer two data points in the design space of effect handlers. Other tradeoffs are possible. More experimentation with different designs and implementations of languages with effect handlers will help to inform these.

A TARGET LANGUAGE OF λ_{Cap} (SIMPLY-TYPED LAMBDA CALCULUS – STLC)

For easier reference, Figure 8 repeats the standard syntax and typing rules of a call-by-value simply-typed lambda calculus [Barendregt 1992] extended with **letrec**.

Syntax of Terms:

$$\text{Expressions} \quad e \;::=\; \mathsf{True} \mid x \mid e \,@\, e \mid \lambda x \Rightarrow e \mid \mathsf{letrec}\, f \;=\; e \;\mathsf{in}\; e$$

Syntax of Types:

$$\text{Types} \quad \tau \;::=\; \tau \to \tau \mid \mathsf{Int} \mid \mathsf{Bool} \mid \ldots$$

$$\text{Type Env.} \quad \Gamma \;::=\; \emptyset \mid \Gamma,\, x : \tau$$

Type Rules:

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

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

Fig. 8. The target language of translating λ_{Cap} – Call-by-value simply-typed lambda calculus with letrec.

B TARGET LANGUAGE OF $\mathbb{\lambda}_{\text{Cap}}$ (TWO-LEVEL LAMBDA CALCULUS)

Figure 9 repeats the standard syntax and typing rules of a 2-level lambda calculus [Danvy et al. 1996; Hillerström et al. 2017; Jones et al. 1993; Nielson and Nielson 1996]. For simplicity, and to be closer to our implementation, we only include residualized constants and only residualized **letrec**.

Syntax of Terms:

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

Syntax of Types:

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

Type Rules:

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

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

Fig. 9. The target language of translating λ_{Cap} – 2-level lambda calculus.

C PROOFS

C.1 Typability of Translated Terms (λ_{Cap})

Our translation preserves typability (Theorem 5.1). To proof this theorem, we have to assume hygiene of our translation. That is, variables in $\mathcal{T}\llbracket\, \Theta \,\rrbracket$ and $\mathcal{T}\llbracket\, \Gamma \,\rrbracket$ 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 $\mathcal{T}\llbracket\, \Gamma \,\rrbracket$. Furthermore, we assume that for every $\mathbb{F}$, there exists a corresponding global signature $\Sigma(\mathbb{F}) = \tau_1 \longrightarrow \tau_2$.

Proof. The proof proceeds by induction on the typing derivation. The cases for the typing judgement $\vdash_{\mathsf{exp}}$ are all straightforward. The most interesting cased in the typing judgement $\vdash_{\mathsf{cap}}$ 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 $\Theta \mid \Gamma \vdash_{\mathsf{cap}} \mathsf{lift}\ h : [\, \mathbb{F} \,]_{\emptyset,\tau}$ we need to show $\mathcal{T}\llbracket\, \Theta \,\rrbracket,\ \mathcal{T}\llbracket\, \Gamma \,\rrbracket \vdash \mathcal{H}\llbracket\, \mathsf{lift}\ h \,\rrbracket : \mathcal{T}\llbracket\, [\, \mathbb{F} \,]_{\emptyset,\tau} \,\rrbracket$.
From the premises, we have $\Theta \mid \Gamma \vdash_{\mathsf{cap}} h : [\, \mathbb{F} \,]_{\emptyset}$ **(1)**. Further, we can compute:
$\mathcal{T}\llbracket\, [\, \mathbb{F} \,]_{\emptyset,\tau} \,\rrbracket = \mathcal{T}\llbracket\, \tau_1 \,\rrbracket \longrightarrow C\llbracket\, \tau_2 \,\rrbracket_{\emptyset,\tau} = \mathcal{T}\llbracket\, \tau_1 \,\rrbracket \longrightarrow (\mathcal{T}\llbracket\, \tau_2 \,\rrbracket \longrightarrow \mathcal{T}\llbracket\, \tau \,\rrbracket) \longrightarrow \mathcal{T}\llbracket\, \tau \,\rrbracket$
This lets us derive:

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

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

Similar to the other case, given $\Theta \mid \Gamma \vdash_{\mathsf{cap}} \mathsf{lift}\ h : [\, \mathbb{F} \,]_{\overline{\tau},\tau,\tau'}$ we need to show
$\mathcal{T}\llbracket\, \Theta \,\rrbracket,\ \mathcal{T}\llbracket\, \Gamma \,\rrbracket \vdash \mathcal{H}\llbracket\, \mathsf{lift}\ h \,\rrbracket : \mathcal{T}\llbracket\, [\, \mathbb{F} \,]_{\overline{\tau},\tau,\tau'} \rrbracket$.
From the premises, we have $\Theta \mid \Gamma \vdash_{\mathsf{cap}} h : [\, \mathbb{F} \,]_{\overline{\tau},\tau}$ **(1)**. Further, we can compute:
$\mathcal{T}\llbracket\, [\, \mathbb{F} \,]_{\overline{\tau},\tau,\tau'} \,\rrbracket = \mathcal{T}\llbracket\, \tau_1 \,\rrbracket \longrightarrow C\llbracket\, \tau_2 \,\rrbracket_{\overline{\tau},\tau,\tau'} = \mathcal{T}\llbracket\, \tau_1 \,\rrbracket \longrightarrow (\mathcal{T}\llbracket\, \tau_2 \,\rrbracket \longrightarrow C\llbracket\, \tau' \,\rrbracket_{\overline{\tau},\tau}) \longrightarrow C\llbracket\, \tau' \,\rrbracket_{\overline{\tau},\tau}$
Again, we can derive

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots\vdash k : \mathcal{T}\llbracket\tau_2\rrbracket\to\mathcal{C}\llbracket\tau'\rrbracket_{\overline{\tau},\tau}}\,\textsc{T-Var} \quad \dfrac{}{\ldots\vdash y : \mathcal{T}\llbracket\tau_2\rrbracket}\,\textsc{T-Var}}{\ldots\vdash k @ y : (\mathcal{T}\llbracket\tau'\rrbracket\to\mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}})\to\mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}}}\,\textsc{T-App} \quad \dfrac{}{\ldots\vdash k' : \mathcal{T}\llbracket\tau'\rrbracket\to\mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}}}\,\textsc{T-Var}}{\ldots, y : \mathcal{T}\llbracket\tau_2\rrbracket \vdash k @ y @ k' : \mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}}}\,\textsc{T-App}}{\ldots\vdash \lambda y \Rightarrow k @ y @ k' : \mathcal{T}\llbracket\tau_2\rrbracket\to\mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}}}\,\textsc{T-Lam}$$

In the derivation, we implicitly expand and contract applications to the meta function $\mathcal{C}\llbracket\cdot\rrbracket$.

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)**, $\mathcal{T}\llbracket [\text{Resume}_i]_{\overline{\tau}}\rrbracket = \mathcal{T}\llbracket\tau''\rrbracket \to \mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}}$, and the ind. hyp., we obtain:
$\mathcal{T}\llbracket\Theta\rrbracket,\ k : \mathcal{T}\llbracket\tau''\rrbracket \to \mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}},\ \mathcal{T}\llbracket\Gamma\rrbracket,\ x : \mathcal{T}\llbracket\tau'\rrbracket \vdash \mathcal{S}\llbracket s\rrbracket_{\overline{\tau}} : \mathcal{C}\llbracket\tau\rrbracket_{\overline{\tau}}$ **(3)**.
Now, starting from $\mathcal{T}\llbracket\Theta\rrbracket, \mathcal{T}\llbracket\Gamma\rrbracket \vdash \lambda x \Rightarrow \lambda k \Rightarrow \mathcal{S}\llbracket s\rrbracket_{\overline{\tau}} : \mathcal{T}\llbracket\tau'\rrbracket \to \mathcal{C}\llbracket\tau''\rrbracket_{\overline{\tau},\tau}$ 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, 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 $\Theta \mid \Gamma \vdash_{\text{cap}} \mathcal{H}\llbracket h\rrbracket_{\overline{\tau},\tau} : [\mathbb{F}]_{\overline{\tau},\tau}$ **(2)**. Following the syntactic abbreviation, we also assume a derived rule T-Let.

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

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 $\ldots x : \mathcal{T}\llbracket\tau\rrbracket \vdash \mathcal{E}\llbracket x\rrbracket : \mathcal{T}\llbracket\tau\rrbracket$.

□

C.2 Typability of Translated Terms ($\lambda\!\!\lambda_{\text{Cap}}$)

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 for λ_{Cap}, we can derive:

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

We apply the typing rules for residualized abstractions T-RLam followed by Lemma 10.1. We use a lemma that the translation of a dynamic type $\mathcal{T}[\![\ \tau\ ]\!]$ 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).

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

REFERENCES

Andrei Alexandrescu. 2010. *The D Programming Language* (1st ed.). Addison-Wesley Professional.

Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 81–89. https://doi.org/10.1145/2889160.2889229

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.

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.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. In *Proceedings of the Symposium on Principles of Programming Languages (to appear)*. ACM, New York, NY, USA.

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

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

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

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

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.

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

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo Bee Doo Bee Doo. *Journal of Functional Programming* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

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

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

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

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

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.

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

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

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

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.

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.

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

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.

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.

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

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

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

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

Oleg Kiselyov and 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. https://doi.org/10.4204/EPTCS.285.2

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.

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

Daan Leijen. 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.

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

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).

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.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the Symposium on Principles of Programming Languages* (San Francisco, California, USA). ACM, New York, NY, USA, 333–343.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the Symposium on Principles of Programming Languages* (Paris, France). ACM, New York, NY, USA, 500–514.

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.

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* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 482–494. https://doi.org/10.1145/3062341.3062380

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

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.

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

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* (Oxford, United Kingdom) *(LICS '18)*. ACM, New York, NY, USA, 809–818. https://doi.org/10.1145/3209108.3209166

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

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

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

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.

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

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.

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

Bjarne Stroustrup. 1997. *The C++ Programming Language, Third Edition* (3rd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation* (Amsterdam, The Netherlands). ACM, New York, NY, USA, 203–217.

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

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

Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free - Efficient Algebraic Effect Handlers. In *Proceedings of the Conference on Mathematics of Program Construction* (Königswinter, Germany). Springer LNCS 9129.

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.