Effect Handlers for the Masses

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

Effect handlers are a program structuring paradigm with rising popularity in the functional programming language community and can express many advanced control flow abstractions. We present the first implementation of effect handlers for Java – an imperative, object oriented programming language. Our framework consists of three core components: A type selective CPS transformation via JVM bytecode transformation, an implementation of delimited continuations on top of the bytecode transformation and finally a library for effect handlers in terms of delimited continuations.

CCS Concepts: • **Software and its engineering → Control structures**; **Source code generation**; **Runtime environments**; **Abstraction, modeling and modularity**;

Additional Key Words and Phrases: effect handlers, algebraic effects, delimited continuations, java, jvm, bytecode transformation

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

1 INTRODUCTION

Algebraic effects [Plotkin and Power 2003] in their extension with effect handlers [Plotkin and Pretnar 2009] are a program structuring paradigm, splitting programs into three parts: (1) *effect signatures*, that declare *effect operations* like for example *yield* to output an element of a (push-based) stream, *getHttp* to send an (potentially asynchronous) http-request, *raise* to throw an exception and so on, (2) *effectful functions*, that call these effect operations either directly or indirectly via other effectful functions, (3) *effect handlers* [Bauer and Pretnar 2013; Plotkin and Pretnar 2009], that implement the effect operations, specifying what it means for example to yield, send http-requests or throw exceptions.

Lacking a general mechanism, many control flow abstractions in Java like generators[1], asynchronous programming with async/await[2], the coroutine programming model[3] and fibers (lightweight user-level threads)[4] are currently implemented by custom source-to-source or bytecode transformations. Since each extension makes different assumptions about the generated code, combining the different concepts in a single project is non-trivial. As has been shown in the literature, using effect handlers, many of these control flow abstractions can be expressed as simple libraries [Dolan et al. 2017, 2015; Leijen 2017b], naturally allowing a combined usage. We will revisit some examples of control flow abstractions in the context of Java in Section 4.

[1]https://github.com/peichhorn/lombok-pg/wiki/Yield
[2]https://github.com/electronicarts/ea-async
[3]https://github.com/offbynull/coroutines
[4]http://docs.paralleluniverse.co/quasar

2475-1421/2018/11-ART111
https://doi.org/10.1145/3276481

Many implementations of effect handlers can be found in functional languages. They are either built into the language, like in Eff [Bauer and Pretnar 2015], Koka [Leijen 2014], Frank [Lindley et al. 2017] and Links [Hillerström et al. 2017], or implemented as a libraries for Haskell [Kammar et al. 2013; Kiselyov et al. 2013; Wu and Schrijvers 2015] or Idris [Brady 2013]. Recently, object-oriented and imperative languages also started to adopt effect handlers, both as a built-in language construct for OCAML [Dolan et al. 2017]), as well as as libraries for OCAML [Kammar et al. 2013; Kiselyov and Sivaramakrishnan 2016], C [Leijen 2017a] and Scala [Brachthäuser and Schuster 2017].

We identify the following important key characteristics of programming languages and libraries with support for effect handlers:

– effectful functions mention the effects they use in their type and an effect type system asserts that all effects are eventually handled;
– effect handlers can be applied locally, describing a dynamic scope in which effect operations of a particular effect signature are handled by this very handler;
– to implement effect operations, handlers gets access to the delimited continuation, that is the remainder of the program up to the enclosing call to the effect handler.

In this paper, we present a framework for programming with effect handlers in Java called Effekt[5]. In our framework, effect signatures are Java interfaces, effect handlers are classes that implement those interfaces and effectful functions use instances of the effect handlers. Our implementation consists of three core components: A type selective CPS transformation via JVM bytecode transformation, an implementation of delimited continuations on top of the bytecode transformation and a library for effect handlers in terms of delimited continuations.

While all three components are designed in concert to implement the effect handler library, they can be used and understood individually. The bytecode transformation for example is performed independent of Java as the source language and could be reused with other JVM languages like Scala, Kotlin, JRuby, Clojure and others.

In short, our contributions are:

– The first library design for programming with effect handlers in Java.
– An implementation of multi-prompt delimited continuations in Java. It uses trampolining and avoids the typical linear overhead of restoring the stack upon resumption common to all continuation libraries in Java that we are aware of.
– A type-selective, signature preserving CPS transformation of JVM bytecode. We use closures introduced in Java 1.8 to create specialized instances of continuation frames. The general idea is applicable to any VM-based language that supports closure creation.
– Examples of how to program with effect handlers in Java.

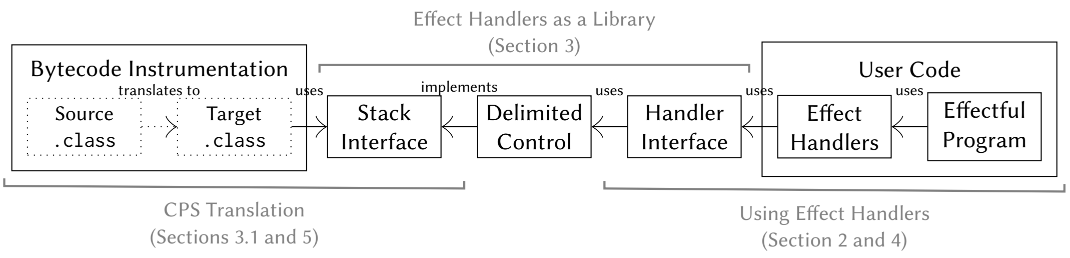

Fig. 1. Structure of the Effekt framework. Directed, solid arrows express dependencies.

[5]https://github.com/double-blind-review/effekt

1.1 Overview

The paper structure relates to the structure of our framework which is presented in Figure 1.

User Code. We show how to program with effect handlers in Java using our library. Section 2 presents the library design of Effekt by means of a standard running example of an effectful program; Section 4 illustrates the expressiveness of Effekt with several more complex examples.

Bytecode Instrumentation. Section 3.1 illustrates the CPS translation by applying it to the running example and section 5 describes the implementation of the translation in more detail. Section 6.3 evaluates the performance overhead induced by our translation and compares the performance of Effekt to other continuation libraries in Java.

Delimited Control. Section 3.2 describes the implementation of multi-prompt delimited continuations [Dyvbig et al. 2007] in Java. It interfaces with the instrumented bytecode by implementing the `Stack` interface that the instrumented bytecode uses. This means that the bytecode instrumentation component can potentially be used as a backend for applications other than our implementation of delimited continuations and that the instrumentation could be exchanged by another backend that implements the `Stack` interface.

Effect Handlers as a Library. Subsection 3.3 shows the (to our knowledge) first implementation of effect handlers as a library for Java. We implement the effect handler library on top of multi-prompt delimited continuations as defined in interface `DelimitedControl`. While the implementation of effect handlers in terms of multi-prompt delimited continuations is not novel [Kiselyov and Sivaramakrishnan 2016], choosing Java as the target language is and poses new challenges like integrating effect handlers with mutable state and designing an API that only uses interfaces and generics for typing. However, it also opens new opportunities like using subtyping and inheritance for handler reuse. The effect handler library is independent of our concrete implementation of `DelimitedControl` and could for instance be reused together with a modified JVM runtime that directly supports delimited continuations. Section 6.2 discusses design decisions in our effect handler library and compares the performance of Effekt with existing effect libraries in Scala.

2 PROGRAMMING WITH EFFECT HANDLERS IN EFFEKT

To get a first impression of how to program with effect handlers in Effekt, let's look at a standard example from the literature [Kammar et al. 2013] – a drunk tossing a coin (Figure 2). In the effectful function *`drunkFlip`* in Subfigure 2a we use the effect operation *`flip`* to nondeterministically choose whether the drunk is too drunk to catch the coin in which case we use the effect operation *`raise`* to raise an exception. Otherwise we return the result of a second coin toss as a string. The two effect operations *`flip`* and *`raise`* are declared in corresponding effect signatures `Amb` (for ambiguity) and `Exc` (for exceptions) in Subfigure 2b.

The type signature of *`drunkFlip`* declares that it may use effects by adding the checked exception `Effects` to its **`throws`** clause. It may use effect operations from the effect signatures `Amb` and `Exc` because it takes instances of those interfaces as arguments. The *`drunkFlip`* method does not rely on any concrete implementation of the effect operations *`raise`* and *`flip`* since it merely uses the interfaces `Amb` and `Exc` in its signature. The caller is free to pick implementations of `Amb` and `Exc`, determining the semantics of the effect operations. For example, we could ignore effect handlers (and our Effekt library) and use real side-effects to implement `Amb` and `Exc` directly. For *`flip`* we use a random number generator and for *`raise`* we use native exceptions.

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  boolean caught = amb.flip();
  if (!caught) {
    return exc.raise("We dropped the coin.");
  } else {
    return amb.flip() ? "Heads" : "Tails";
  }
}
```

(a) Effectful function modeling a drunk trying to flip a coin.

```
interface Exc {
  <A> A raise(String msg) throws Effects;
}
interface Amb {
  boolean flip() throws Effects;
}
```

(b) Effect signatures for Exc and Amb.

Fig. 2. Example of using two effects in an effectful function.

To run our example function, we pass instances of *NativeExceptions* and *RandomFlip* to *drunkFlip* which will randomly result in either a runtime exception or one of the two possible string values.

```
class NativeExceptions implements Exc {
  <A> A raise(String msg) throws Effects {
    throw new RuntimeException(msg);
  }
}
```

```
class RandomFlip implements Amb {
  boolean flip() throws Effects {
    return Math.random() > 0.5;
  }
}
```

The interpretations using native side-effects are not the only ones possible. In the remainder of this section we will explore an interpretation of programs that use Exc into programs that return an Optional[6]. Calling *raise* will immediately abort the program with Optional.*empty*(). We will also explore an interpretation of programs that use *flip* into programs that return a list by enumerating and collecting all possible outcomes. Note that these interpretations are not type-preserving: They change the result type of each function that uses Exc to return an optional and each function that uses Amb to return a list. This means that, without support for effect handlers, we would have to change how these results are processed by every caller of an effectful function. What is even worse, if we use both effects together (as in *drunkFlip*) we would need to decide and fix upfront whether raising an exception terminates the search for possible outcomes and thus the function returns Optional<List<*R*>>, or whether it only terminates one branch in our search and thus our function returns List<Optional<*R*>>. Revising decisions like this in retrospect can be very costly since they potentially affect the whole codebase. If we want to use functions that use only one of Exc and Amb together with functions that use both we have to manually "lift" the functions that use only one of the two effects. Finally, in interpreting the two effect operations together, we cannot immediately reuse the implementations of the individual effect operations. Effect handlers address these problems.

[6] Optional<*A*> is an interface for optional values of type *A*, introduced in Java 1.8

2.1 Handling Effects

We now show how to freely mix handlers, if we use effect operations like in Figure 2 and implement handlers for them by using our Effekt library. This is an example of the modularity benefits provided by effect handlers. The full implementation of our handlers will be given shortly, for now it is enough to know that we define our handlers as follows:

```
class Maybe<R>   extends Handler<R, Optional<R>> implements Exc  {...}
class AmbList<R> extends Handler<R, List<R>>     implements Amb {...}
```

Like our first interpretation of Amb and Exc above, the classes Maybe and AmbList again implement the corresponding effect signatures. The handlers extend our library class Handler<*R*, *E*> to express that they represent effect handlers. In general, every handler for an effect gives semantics to the corresponding effect operations. It interprets an effectful program which would compute a result of type *R* (mnemonic for "return type") into a new semantic domain of type *E*, the *effect domain*. The effect domain is chosen to have enough structure to implement the effect operations. Before looking at the details of how the handlers are implemented, it is instructive to understand how they can be used. This is best explained by analogy to exception handling. Effect handlers are a generalization of exception handlers [Plotkin and Pretnar 2009]: Effect operations generalize **throw** to other effects while handling an effect operation is a generalization of **try** {...} **catch** {...}. In pseudo syntax, handling effect operations like *flip* and *raise* can be thought of as:

```
try { try { ... flip()...raise("...") ... } catch Exc with Maybe} catch Amb with AmbList
```

Like with exception handlers in Java, the try-block describes the dynamic scope of the corresponding handler. In comparison, using Effekt, handling effects Amb and Exc looks like:

```
List<Optional<String>> res1 = handle(new AmbList<Optional<String>>(), amb →
                                  handle(new Maybe<String>(), exc → drunkFlip(amb, exc)));
```

The lambda, which is passed as body to handle(*h*, *body*) represents the dynamic scope in which the handler *h* can be used. Besides registering the handler *h* for the dynamic scope, handle also passes the handler unmodified as argument to the body. Thus, in our example the variable *amb* will be bound to the instance of AmbList created on the same line. Running *drunkFlip* with both effects handled now yields for res_1:

```
▶ [Optional["Heads"], Optional["Tails"], Optional.empty]
```

We can easily swap the two handlers to obtain a different semantics where an exception leads to a termination of the backtracking search.

```
Optional<List<String>> res2 = handle(new Maybe<List<String>>(), exc →
                                  handle(new AmbList<String>(), amb → drunkFlip(amb, exc)));
```

Since at least one of the control paths raises an exception, running *drunkFlip* with the effects handled in the different order yields for res_2:

```
▶ Optional.empty
```

This shows the power of effect handlers: effectful programs can be written fully agnostic of both the semantic domain into which the effects will be interpreted, as well as the order in which the effects will be handled.

2.2 Implementing Effect Handlers

Having seen how effects can be handled, we will now take a closer look at how handlers are implemented. Subfigure 3b shows the Handler interface which is relevant for implementing effect handlers. It also shows the type of effectful functions `Eff<S, T>` which is just like the Java function interface `Function<S, T>` but with its single abstract method being marked as throwing `Effects`. Similarly, interface `CPS<A, E>` corresponds to the nested effectful function type `Eff<Eff<A, E>, E>`.

```
class Maybe<R> extends Handler<R, Optional<R>>
    implements Exc {
  Optional<R> pure(R r) { return Optional.of(r); }
  <A> A raise(String msg) throws Effects {
    return use(k → Optional.empty());
  }
}
class AmbList<R> extends Handler<R, List<R>>
    implements Amb {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k →
      Lists.concat(k.resume(true),
                   k.resume(false)));
  }
}
```

(a) The two effect handlers `Maybe` and `AmbList` utilizing `use` to capture the continuation.

```
abstract class Handler<R, E> {
  E pure(R r) throws Effects;
  <A> A use(CPS<A, E> body) throws Effects {
    ...
  }
  static <R, E, H extends Handler<R, E>> E ↩
    handle(H h, Eff<H, R> p) throws Effects {
    ...
  }
}
interface Eff<S, T> {
  T resume(S value) throws Effects;
}
interface CPS<A, E> {
  E apply(Eff<A, E> k) throws Effects;
}
```

(b) Interface of the library class `Handler` and the necessary functional interfaces.

Fig. 3. Implementation of effect handlers for `Exc` and `Amb` using the library class `Handler`.

Every handler needs to implement the abstract method *pure* to specify how handled programs that don't use the corresponding effect are lifted from R into the effect domain E. Additionally, a handler needs to implement all the effect operations which are specified in the effect signature. To implement the effect operations, the handlers are able to utilize the instance method `use` provided by the library class `Handler`. Calling `this.use(body)` in the implementation of an effect operation captures the continuation k and passes it to the provided body. It then continues executing *body*(k), effectively passing control to the handler. This allows the handler to suspend and resume the handled effectful program.

Both handlers, `Maybe` and `AmbList` implement their effect operations in terms of `use` (Figure 3a). The handler `Maybe`, captures the continuation and deliberately discards it in order to implement the *raise* effect. This mimics the behavior of native exceptions, where an exception causes the unwinding (and discarding) of the runtime stack up to the corresponding exception "handler" (hence the terminology). The handler for ambiguity, in turn, captures the continuation and invokes it twice. Once with **true** and once with **false**, each yielding a list of possible results. Finally, it concatenates the two lists[7]. It is important to stress that this is only possible because the continuation captured by `use` is delimited by the corresponding call to `handle`. In the implementation of *flip*, the continuation captured by `use` will return a list because it is delimited by the call to `handle(new AmbList<String>(), ...)` and the handler `AmbList` defines

[7] For this example, we assume lists to be immutable and only be constructed by *singleton* and *concat*.

the effect domain to have the type List<String>. In general, the type of the captured continuation is Eff<*A*, *E*>. That is, it is an effectful function from *A* to *E*. Similarly, in *raise* it is safe to discard the continuation and immediately return Optional.*empty* because the caller of handle(**new** Maybe<String, Optional<String≫(), ...) expects a value of type Optional<String>.

2.3 Design of the Effekt Library

We designed Effekt as a library for Java around object oriented idioms like subclassing and dynamic dispatch. In summary, the Effekt library is based on the following design decisions. Effect signatures are interfaces and handlers are classes implementing the interfaces. We establish a *handler passing style* where users explicitly pass handler instances to functions that use effect operations. As we discuss in Section 6, this corresponds to a shallow embedding of effect handlers and is similar to the use of type classes by Kammar et al. [2013] and implicits by Brachthäuser and Schuster [2017]. Handler implementations capture the continuation by explicitly calling use. In Effekt, *tail resumptive* handler implementations which exclusively invoke the continuation in tail position can be expressed as simple methods that do not capture the continuation. Effekt is a library and neither changes the language nor the type system. Effectful functions always need to be marked to throw an Effects exception, since only marked methods are transformed by our CPS transformation. The exception should never be caught or otherwise suppressed. Handlers and the operations use and handle are designed to not require an advanced type system. In particular, they do not require type constructor polymorphism. Effekt does not establish a full static effect typing discipline. Users need to make sure that an effect handler is only used in the dynamic extent of a corresponding call to handle. Calling use directly, or indirectly via some effect operation on a handler outside of the handler scope will result in a runtime error.

3 IMPLEMENTING AN EFFECT HANDLER LIBRARY IN JAVA IN THREE STEPS

As can be seen from our running example, programming with effect handlers in Effekt is almost just standard Java programming. Only the control operator use and its counterpart handle make the difference in expressivity. This section describes how these control operators can be implemented, bottom up. We start with a CPS translation, that rewrites all methods annotated with **throws** Effects, build a library for delimited continuations upon the translation and finally implement effect handlers in terms of delimited continuations.

3.1 Type Selective CPS Transformation by Example

To support accessing the continuation with use, the Effekt framework performs a type selective CPS transformation by instrumenting (that is, rewriting) JVM bytecode. This can either be achieved by hooking into the class loading mechanisms of Java and injecting the transformation at runtime when a class is loaded or by a separate preprocessing phase that rewrites the class files once ("ahead of time"). Implementing the transformation on the level of JVM bytecode opens up the opportunity of reuse for other JVM languages.

While the implementation of Effekt rewrites JVM bytecode, for easier accessibility of the paper this section presents the CPS transformation as a semantically equivalent[8] source-to-source rewriting of the example program *drunkFlip*. This section provides an overview, Section 5 describes the implementation of the bytecode transformation in more detail and explains how we treat control flow and exceptions.

[8]For the example presented in this section, we manually verified that the bytecode of the source-to-source transformation is equivalent to the result of the bytecode transformation (modulo some superfluous register stores/loads).

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  boolean caught = amb.flip() ;
  if (!caught) {
    return exc.raise("We dropped the coin.") ;
  } else {
    return amb.flip() ? "Heads" : "Tails";
  }
}
```

(a) Source method with highlighted effect calls. The effect call to *exc.raise* is a tail call and does not require an entrypoint.

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  Effekt.push(() → drunkFlip0(amb, exc));
  return null;
}
```

(b) Generated method stub, only pushing the initial entrypoint.

```
static void drunkFlip0(Amb amb, Exc exc) {
  Effekt.push(() → drunkFlip1(amb, exc));
  amb.flip() ;
}
static void drunkFlip1(Amb amb, Exc exc) {
  boolean caught = Effekt.result();
  if (!caught) {
    exc.raise("We dropped the coin.") ;
  } else {
    Effekt.push(() → drunkFlip2(amb, exc, caught));
    amb.flip() ;
  }
}
static void drunkFlip2(Amb amb, Exc exc, ↩
    boolean caught) {
  boolean res$1 = Effekt.result();
  Effekt.returnWith(res$1 ? "Heads" : "Tails");
}
```

(c) Entrypoints as separate, static method.

Fig. 4. CPS translation of the example in Figure 4a, presented as a source-to-source transformation.

Figures 4b and 4c show the result of transforming the method *drunkFlip*. We instrument only effectful methods and identify those by means of the **throws** Effects annotation. Using Reynolds [1972] terminology, we only consider methods marked with **throws** Effects to be "serious". All other functions are "trivial" and don't require any instrumentation. Consequently, we also only instrument call sites of effectful functions (*effect calls*). In *drunkFlip* there are three such effect calls, two to *flip* and one to *raise*. We exclude *tail effect calls* from the translation, that is effect calls immediately followed by a return. For *drunkFlip* this means that we instrument the two *flip* calls, since the call to *raise* is in tail position. We call the code immediately following an effect call an *entrypoint*. We also treat the initial entrypoint of a function as an entrypoint in this sense.

Similar to Prokopec and Liu [2018], for each entrypoint in an effectful method, we generate a separate *entrypoint method*. For our example, these are the methods $drunkFlip_0$ (the initial entrypoint method), $drunkFlip_1$ and $drunkFlip_2$ (corresponding to the two invocations of *flip*). Entrypoint methods take the *function local state* as arguments. That is, all local variables and values on the operand stack needed to resume the function execution after the effect call would return.

Similarly to how the JVM would push a stack frame before a method call, we rewrite every effect call to first push a *continuation frame* to a global user-level stack by invoking Effekt.*push*. Class Effekt has a global static field Effekt.*stack* that implements the interface Stack shown in Figure 5. For notational convenience, we write Effekt.*push* instead of Effekt.*stack*.*push*. The interface Stack contains all necessary methods used by the instrumented bytecode. A continuation frame is an instance of the Frame interface, also shown in Figure 5. We use Java 8 lambdas to create instances of the Frame interface. The lambdas close over the function local state which they will pass to the entrypoint methods when invoked with *enter*. Conceptually, we thus represent continuation frames as instances of classes that have one field for each local they store. After pushing the continuation frame, the entrypoint methods call the effectful method and immediately return. Thus, all effect calls in the translated program are tail calls.

```
interface Stack {
  // special calling convention
  void returnWith(Object r);
  void unwindWith(Throwable t);
  <A> A result() throws Throwable;
  // stack of frames
  void push(Frame frame);
  Frame pop();
  boolean isEmpty();
}

interface Frame {
  void enter();
}
```

```
abstract class RTStack implements Stack {
  Object res; Throwable exc;
  void returnWith(Object r) { res = r; exc = null; }
  void unwindWith(Throwable t) { res = null; exc = t; }
  <A> A result() throws Throwable {
    if (exc ≠ null) throw exc;
    return (A) res;
  }
  void trampoline() {
    while (!isEmpty())
      try { pop().enter(); }
      catch (Throwable t) { unwindWith(t); }
  }
}
```

Fig. 5. Interface and example implementation of the user-level stack.

Instrumented effectful functions use a special calling convention: We rewrite all returns to Effekt.*returnWith* calls. Correspondingly, entrypoint methods use Effekt.*result* to obtain the result of the previous effect call. We transform the original method *drunkFlip* to a stub (Figure 4b) that pushes a continuation frame for the *initial entrypoint* and immediately returns a dummy value. Callers of *drunkFlip* have to use Effekt.*result* to eventually get the actual return value. Instances of Frame follow the same calling convention for consistency, as we can observe in $drunkFlip_2$.

The design of the CPS transformation has been guided by the goal to maximize interoperability with non-effectful functions, such as library functions that do not use effect operations and thus are not instrumented. At the same time we aimed to support interoperability with other Java features with as little specialization of the transformation as possible. This includes interfaces, separate compilation, generics, dynamic method dispatch, subtyping, lambda expressions, visibility modifiers, nested classes and native exceptions. The most important consequence of these design goals is that our CPS translation preserves method signatures and thus does only translate terms, not types or signatures. A standard CPS translation changes the type of a computation that returns A to a function type $(A \longrightarrow R) \longrightarrow R$ for some answer type R. However, this change is precluded by our design decision of not changing method signatures. Instead, the continuation $A \longrightarrow R$ is obtained via the global instance of Stack, as we will see in the next subsection. To accommodate for the change in return type, we make effectful functions use our own custom calling convention.

Instead of using the JVM stack for continuation frames, we use a separate user-level stack. A canonical implementation of Stack recovering the expected runtime behavior of the JVM stack is sketched as class *RTStack* in Figure 5. Only the methods implementing our calling convention are provided, straightforward stack operations *push*, *pop*, and *isEmpty* are left abstract for lack of space. Our stack implementation performs trampolining [Ganz et al. 1999]. To run an instrumented function we first invoke it to push a frame that corresponds to its initial entrypoint onto Effekt.*stack*:

```
static <A> A run(Eff<Void, A> prog) {
  prog.resume(null);
  Effekt.trampoline();
  return Effekt.result();
}
```

```
Effekt.run(() → handle(new AmbList <>(), amb →
                  handle(new Maybe <>(), exc →
                    drunkFlip(amb, exc))));
```

To actually start execution, we call `Effekt.trampoline` which will continue to pop and enter frames until the stack is empty. If an exception is raised, the trampoline will unwind the user-level stack frame by frame. Each frame starts with a call to *result*, reraising the exception after restoring the method state. Section 5 shows more details on how we deal with exceptions.

3.2 Delimited Continuations

A program that has been transformed with our CPS translation still uses the JVM stack for non-effectful calls but uses our user-level stack for effectful function calls. Different implementations of the interface `Stack` with different representations also give rise to different additional operations that exploit the corresponding stack representation. In consequence, effectful programs that are executed against a particular stack implementation can make use of those additional operations. In this section, we will develop one particular implementation of `Stack` that implements additional operations to capture delimited continuations. All code in the rest of this paper is subject to the CPS bytecode transformation and all methods annotated with **throws** Effects will be instrumented.

As we will see in subsection 3.3, the effect handler library can be implemented as a very thin layer on top of multi-prompt delimited continuations [Brachthäuser and Schuster 2017; Kiselyov and Sivaramakrishnan 2016]. Our implementation of multi-prompt delimited continuations closely follows Dyvbig et al. [2007], but translated to Java and to our setting of bytecode instrumentation. We extend the `Stack` implementation sketched above and implement two additional methods *pushPrompt* and *withSubcont* as summarized by the following interface:

```
interface DelimitedControl {
  <E>     E pushPrompt(Prompt<E> p, Eff<Void, E> prog) throws Effects;
  <A, E>  A withSubcont(Prompt<E> p, CPS<A, E> body) throws Effects;
}
```

Instances of `Prompt<E>` (interface defined in Figure 6c) are used to mark positions on the stack. The type of an effectful function `Eff` and of effectful programs that use an effectful continuation `CPS` have been defined in Figure 3b.[9] The type parameter E of `Prompt` unifies with the type of the computation that we delimit with *pushPrompt*. Capturing a continuation with *withSubcont*, the return type of the continuation and of the body have to match the type of the prompt E.

3.2.1 Using Delimited Continuations. Assuming that the global stack instance supports the methods from `DelimitedControl`, for any two different prompts p_1, p_2 : Prompt<Integer> we can implement examples such as:

$2 * \text{Effekt}.pushPrompt(p_1,\ () \rightarrow 1 + \text{Effekt}.withSubcont(p_1, k \rightarrow k.resume(2) * k.resume(6)));$

▶ $2 * ((1 + 2) * (1 + 6)) = 42$

The continuation k corresponds to the evaluation context $1 + \square$, since it is delimited by p_1. We invoke it twice. A second example illustrates prompt search and discarding of the continuation:

$2 * \text{Effekt}.pushPrompt(p_1,\ () \rightarrow 1 + \text{Effekt}.pushPrompt(p_2,\ () \rightarrow 3 * \text{Effekt}.withSubcont(p_1, k \rightarrow 21)));$

▶ $2 * 21 = 42$

The captured continuation k contains the program segment marked by prompt p_1. It corresponds to the evaluation context $1 + pushPrompt(p_2,\ () \rightarrow 3 * \square))$. We discard the continuation and replace it by the value 21.

[9] When effectful functions don't require an argument we will use Eff<Void, A>. To avoid materializing instances of Void and binding them, we write $f.resume()$ as a short hand for $f.resume(\textbf{null})$ and $(() \rightarrow \ldots)$ instead of $(unusedVoid \rightarrow \ldots)$.

```
class SeqStack extends RTStack {
  Seq<Frame> s = Seq.empty();
  void push(Frame frame) { s = s.push(frame); }
  Frame pop() { Frame f = s.head(); s = s.tail(); return f; }
  boolean isEmpty() { return s.isEmpty(); }
}
```

(a) Implementation of Stack, forwarding to an immutable stack.

```
class DelimCC extends SeqStack implements DelimitedControl {
  <E> E pushPrompt(Prompt<E> p, Eff<Void, E> prog)
      throws Effects {
    s = s.mark(p) ; return prog.resume();
  }
  <A, E> A withSubcont(Prompt<E> p, CPS<A, E> body)
      throws Effects {
    Seq<Frame> init = s.before(p); s = s.after(p);
    Eff<A, E> k = (A value) → {
      s = init.prependTo(s) ; return (E) value; }
    return (A) body.apply(k);
  }
}
```

(b) Implementation of delimited control in terms of an immutable, splittable stack: Seq. Usage of Seq is *highlighted*.

```
interface Prompt<E>{ }
```

(c) Interface Prompt, used to mark stack segments. Prompts are compared with reference equality.

```
interface Seq<A> {
  boolean   isEmpty();
  A         head();
  Seq<A>    tail();
  Seq<A>    before(Prompt<?> p);
  Seq<A>    after(Prompt<?> p);
  Seq<A>    push(A element);
  Seq<A>    mark(Prompt<?> p);
  Seq<A>    prependTo(Seq<A> init);
  static <A> Seq<A> empty() {...}
}
```

(d) Immutable stack that allows marking and splitting at positions p.

Fig. 6. Implementation of control operators to capture delimited continuations.

3.2.2 *A Splittable Stack Implementation.*

The simplest implementation of Stack that comes to mind is to store a list of frames in a field s and implement all abstract operations of Stack by forwarding to this list s. Figure 6a drafts such an implementation. For now we just assume Seq to be an immutable implementation of a stack data structure with elements of type A. While very simple, running a program with this stack implementation already has the benefit that it performs trampolining and thus reduces JVM stack usage, which might avoid stack overflows. However, the real power of the translation comes from the fact that Stack implementations can add new methods which expose additional (control) operators. To implement the additional control operators *pushPrompt* and *withSubcont* we need to mark positions on the runtime stack (*mark*), slice the stack at given positions (*before*, *after*) and prepend whole stack segments (*prependTo*) [Dyvbig et al. 2007]. The stack data structure Seq (Figure 6d) that we have already used above offers exactly these operations. One can think of Seq<A> as a two-sorted stack that contains elements of type A and markers of type Prompt. Calling $s.before(p)$ returns the initial segment up to, but not including the first occurrence of the marker p. This segment contains all the recently pushed elements after p has been pushed. Calling $s.after(p)$ returns the remainder of the stack. This segment contains all the elements which have been pushed before the marker p has been pushed. This is characterized by the following equation:

$$s.before(p).prependTo(s.after(p).mark(p)) \equiv s \quad \textbf{where } p \notin s.before(p)$$

3.2.3 Pushing a Prompt. Figure 6b shows the implementation of the DelimitedControl interface, using Seq. To implement *pushPrompt*, we mark the stack using the provided prompt p and update the mutable reference s with the now marked stack. We then resume with the effectful program *prog*. Being effectful, the program *prog* pushes additional frames onto the stack. We can capture those frames later by slicing the stack at the position of the installed marker p.

3.2.4 Capturing a Continuation. The control operator *withSubcont*$(p, body)$ captures the continuation k up to the next dynamically enclosing *pushPrompt*$(p, \ldots)$ and conceptually replaces the call to *pushPrompt* with a call to *body*.*apply*(k). Its implementation in Figure 6b stores all frames that have been pushed after p in a local variable *init*. This segment corresponds to the delimited continuation from type A to type E. That is, the top most frame expects Effekt.*result* to return a value of type A. The initial segment and the prompt marker are then removed from the stack by mutating it with $s = s.after(p)$. This leaves a segment on the stack which expects a value of type E to continue program execution. The continuation k implements the functional interface Eff with method *resume*() **throws** Effects. It closes over the initial stack segment *init* and, when invoked, prepends it to the stack s. This implements the desired semantics of resuming the delimited continuation: The runtime system will first run the initial stack segment *init* before it eventually continues at the callsite of *resume* within *body*.

There are two casts involved, that require some explanation. Both *withSubcont* and the continuation k are effectful. Thus the respective caller will be instrumented. However, by mutating field s and modifying the stack, we change the execution context. In *withSubcont* we remove the initial segment of the stack and thus the new caller expects a value of type E not A. In the continuation we prepend the initial segment and thus the caller now expects an A not E. The Java typechecker is ignorant of our transformation and the modifications to our own callstack. Hence the casts[10].

3.3 Implementation of the Effekt Library

In the previous subsections we have seen how programs which contain **throws** Effects annotations are CPS translated. We extended the runtime environment in which those translated programs are executed to support multi-prompt delimited continuations. Equipped with multi-prompt delimited continuations we are now finally ready to see how the effect handler library can be implemented.

The expressive power of effect handlers comes from the two operations handle and use which are encapsulated in the library class Handler. Figure 7 shows the implementation of these two operations in terms of delimited continuations. We use handlers themselves as prompt markers. An effect handler with effect domain E implements Prompt<E>. The answer type of delimited continuations thus will be the effect domain E. In the implementation of handle we push the handler as a prompt before resuming with *prog*. The pushed handler will delimit the extent of continuations captured by that handler. By calling *h.pure* after resuming, programs that don't use effects will be lifted from R to E. The method use calls *withSubcont* with the current handler instance **this** as a prompt marker to capture the continuation up to the most recent call to handle on this handler instance. It then passes the captured continuation k to *body*.

Effect handlers in Effekt are *deep handlers* [Kammar et al. 2013]. That is, all effect operations are recursively handled by the same handler. To implement deep handlers, we modify the continuation to re-push the prompt before resuming to make sure that all subsequent calls to use on this handler are again delimited by **this**. That is, both continuations k_1 and k_2 in handle$(h, () \rightarrow 1 + h.use(k_1 \rightarrow \ldots) + h.use(k_2 \rightarrow \ldots))$ should be delimited by handle$(h, () \rightarrow \square)$. Our operations handle and use are thus conceptually very similar to *spawn* and the corresponding *controller* by Hieb and Dybvig [1990].

[10]Since A and E are generic type parameters, they will nevertheless be erased and the program can safely be executed.

```
abstract class Handler<R, E> implements Prompt<E> {
  E pure(R r) throws Effects;
  <A> A use(CPS<A, E> body) throws Effects {
    return Effekt.withSubcont(this, k → body.apply(a → Effekt.pushPrompt(this, () → k.resume(a))));
  }
  static <R, E, H extends Handler<R, E>> E handle(H h, Eff<H, R> prog) throws Effects {
    return Effekt.pushPrompt(h,() → h.pure(prog.resume(h)));
  }
}
```

Fig. 7. The essence of the effect handler library: The Handler class.

4 USE CASES

Having implemented effect handlers as a library for Java, programmers can now freely combine object oriented Java programming with effect handlers. We show some use cases that demonstrate this newly gained expressiveness of combining effect handlers with OOP. At the same time we show more of our programming model and how it interacts with OO features such as interfaces, inheritance, etc.

4.1 Handling Multiple Effects in one Handler

All handlers we have seen so far only implemented a single effect signature. However, sometimes it is necessary to group the implementation of multiple effect signatures in a single handler. Since effect signatures are interfaces and handlers are classes implementing those interfaces, this is straightforward. Effect implementations grouped in a single handler share the same effect domain *E*, share the private state of the handler and they can be implemented in terms of each other. In particular, sharing the effect domain is important if the handler wants to express interaction between different effect operations. Two examples combining Amb and Exc in one handler to share the effect domain are:

```
class Nondet<R> extends AmbList<R> ↩
    implements Exc {
  <A> A raise(String msg) throws Effects {
    return use(k → Lists.empty());
  }
}
```

```
class Backtrack<R> extends Maybe<R> ↩
    implements Amb {
  boolean flip() throws Effects {
    return use(k → {
      Optional<R> res = k.resume(true);
      return res.isPresent() ? res : k.resume(false); });
  }
}
```

Handler Nondet extends AmbList and only provides the definition for *raise*. It shares the effect domain List<*R*> with AmbList. Similarly, handler Backtrack extends Maybe and adds the implementation for *flip*. It shares the effect domain Optional<*R*> with Maybe. By subtyping, the combined handlers can of course still be used as handlers for the individual effects Amb or Exc:

```
handle(new Nondet, nd → drunkFlip(nd, nd))
▶ ["Heads", "Tails"]
```

```
handle(new Backtrack, bt → drunkFlip(bt, bt))
▶ Optional["Heads"]
```

This also illustrates reuse of handler implementations by inheritance. We only needed to provide the missing definitions, all other implementations of effect operations are inherited.

4.2 Alternatives to Handler Passing

In previous examples, effectful functions expressed their use of effects by expecting handler instances as arguments, which we refer to as *handler passing style*. In an object oriented programming language like Java, it is natural to explore other means to get access to a handler instance.
Handlers can be passed to constructors and stored in fields. Take the following implementation of a reader effect (specialized to characters) as an example:

```
class StringReader<R> extends Handler<R, R> implements Reader {
  final Exc exc;  final String input;  int pos = 0;

  StringReader(String s, Exc e ) { this.input = s; this.exc = e; }

  public char read() throws Effects {
    return  if (pos ⩾ input.length()) exc.raise("EOS") ;
               else input.charAt(pos ++);
  }
}
```

```
interface Reader {
  char read() throws Effects;
}
```

The handler StringReader uses an instance of an Exc -handler to raise the end-of-stream exception. This instance is passed on construction and stored in the field *exc*. The methods of StringReader which use the Exc effect can only be safely executed in the corresponding dynamic scope of the Exc -handler, which is up to the user of StringReader to ensure.
Under the assumption that the whole program runs in the dynamic scope of one global Exc handler, we could also store this handler instance in a global static field. This has the disadvantage that neither the method signature nor the constructor indicate the use of the Exc effect.
Dependencies on other effect operations can also be expressed by declaring them as abstract methods or abstractly implementing corresponding effect signatures.

4.3 Ambient State and Parametrized Handlers

Our bytecode instrumentation only saves and restores function local state, but does not deep-copy heap allocated state, like fields. Nothing prevents the user from using mutable state in the implementation of handlers. The handler StringReader is an example since it mutates the field *pos*. However, mutable state and delimited continuations can interact in unforeseen ways. For instance, we could use both Amb and Reader in a program *p* and run the program with:

```
handle(new AmbList <>(), amb → handle(new StringReader <>(), rd → p(amb, rd)));
```

As we recall, the AmbList handler resumes a captured continuation twice. Now the first resumption can potentially affect the result of the second resumption by mutation of the field *pos* in the handler StringReader . Instead, for the second resumption we want to reset all handlers that are part of the captured continuation to the state prior to the first resumption. This notion of handler state is referred to as *ambient state* [Leijen 2017b]. In Effekt we support ambient state as follows: Handler implementations can just use mutable state, like StringReader . To turn the handler state into ambient state a handler just needs to implement the Stateful<*S*> interface below.

```
class StringReader₂<R> extends StringReader<R> ↩
    implements Stateful<Integer> {
  Integer exportState() { return  pos; }
  void importState(Integer state) { pos = state; }
}
```

```
interface Stateful<S> {
  S exportState();
  void importState(S state);
}
```

Our implementation of delimited control is extended to save the handler state on capture of the continuation and restore it on resumption. This implements ambiently scoped state.

4.4 Case Study: Parsing

Equipped with nondeterministic choice, exceptions and reader we can implement parsers [Leijen 2016]. For convenience, we group Amb, Exc, and Reader into one effect signature for parsers. As can be seen from the example, *flip* models alternatives in a grammar.

```
interface Parser extends Amb, Exc, Reader { }

int digit(Parser p) throws Effects {
  char t = p.read();
  return  isDigit(t) ? getNumericValue(t)
                     : p.raise("...");
}
```

```
int number(Parser p) throws Effects {
  int res = digit(p);
  while (true)
    if (p.flip()) { res = res * 10 + digit(p); }
    else { return  res; }
}
```

We parse the string "123" with the *number* parser, handling effects with a ForwardingParser that stores effect handlers for Amb, Exc in Reader in fields and implements the corresponding effect operations by forwarding. We can handle effect operations *flip* and *raise* with either the Nondet handler to get a list of all possible parses or with the Backtrack handler to get just the first successful parse if it exists.

```
handle(new Nondet <>(), nd →
  handle(new StringReader₂ <>("123", nd), r →
    number(new ForwardingParser(nd, nd, r))));
▶  ["123", "12", "1"]
```

```
handle(new Backtrack <>(), bt →
  handle(new StringReader₂ <>("123", bt), r →
    number(new ForwardingParser(bt, bt, r))));
▶  Optional["123"]
```

By using effect operations, the parsers can be written in direct style. At the same time handlers (like AmbList or Nondet) can transparently access the continuation in the implementation of effect operations. Similarly, an alternative handler implementation of Reader could access the continuation and convert the parser implementation from pull to push. This way, alternative parsing strategies such as breadth first parsing and online parsing can be implemented without needing to change concrete parsers, like *number*.

4.5 Case Study: Generators and Coroutines

In the programming language Python, the built-in control operation *yield* can be used to describe a stream of values also known as generators [Politz et al. 2013]. While generators are built-in into Python, with effect handlers we can implement them as a library [Leijen 2016].

```
void numbers(int to, Writer<Integer>  w) throws Effects {
  int n = 0;
  while (n ⩽ to) { w.write(n ++); }
}
class Iterate<A, R>  extends Handler<R, Iterator_Eff<A>>
   implements Writer<A>  {...}
```

```
interface Writer<A>  {
  void write(A value) throws Effects;
}
interface Iterator_Eff<A>  {
  boolean hasNext() throws Effects;
  A next() throws Effects;
}
```

The method *numbers* describes a generator that yields integers up to a given value using the Writer effect. We can handle the writer effect with the Iterate handler, which captures and stores the continuation on every write to suspend the generator until the next value is requested.

```
Iterator_Eff<Integer> it = handle(new Iterate()<>, writer → numbers(10, writer));
while (it.hasNext()) { println(it.next()); }
```

Since the iterator is effectful, we cannot reuse the Java interface *Iterator*. The interface $\text{Iterator}_{\mathit{Eff}}$ duplicates the interface much like we introduced Eff as effectful alternative to Function.

The programming model of coroutines [Conway 1963; Prokopec and Liu 2018] is in essence very similar to generators. Coroutines can suspend themselves by yielding values to the coroutine caller, similar to how the Writer effect allows to send values from the generator to the program that uses it. We can easily generalize the Writer effect to allow bidirectional communication to allow the coroutine caller to send values to the coroutine on resumption.

```
interface Yield<A, B> {
  B yield(A value) throws Effects;
}
int coroutine_1(Yield<Integer, Boolean> y) throws Effects {
  int n = 0; while (y.yield(n)) n ++; return n;
}
```

```
interface Coroutine<A, B, R>{
  void resume(B value) throws Effects;
  boolean isDone();
  A value();
  R result();
}
```

In our model, coroutines like $\mathit{coroutine}_1$ are effectful programs using the Yield effect.

```
Coroutine<Integer, Boolean, Integer> c = Coroutine.call(this :: coroutine_1);
System.out.println(co.value());   // 0
co.resume(true);
System.out.println(co.value());   // 1
co.resume(false);
System.out.println(co.result());  // 1
```

In the above example, the coroutine $\mathit{coroutine}_1$ suspends execution after yielding an integer, awaiting a boolean from the coroutine caller to indicate whether further values should be produced. The coroutine instance is created by handling the yield effect with a handler that captures and stores the continuation upon *yield*, similar to how the Iterate handler handles the *write* effect operation.

```
class RoundRobin extends Handler implements Coop {
  Queue<Eff<Void, Void>> ps
  void fork(Eff<Void, Void> p) throws Effects {
    if (flip()) { p.resume(); use(k → null); }
  }
  void yield() throws Effects { use(k → ps.add(k)); }
  boolean flip() throws Effects {
    return use(k → { ps.add(() → k.resume(true));
                     return k.resume(false); });
  }
  void run() throws Effects {
    while (!ps.isEmpty()) ps.remove().resume();
  }
}
```

```
interface Coop {
  void fork(Eff<Void, Void> p) throws Effects;
  void yield() throws Effects;
}
// program using cooperative multitasking
RoundRobin scheduler = new RoundRobin()
handle(scheduler, s → {
  s.fork(() → {
    println("world");
  });
  println("hello");
  s.yield();
});
scheduler.run();
```

Fig. 8. Effect signature Coop with operations for cooperative multitasking and a round-robin scheduler implemented as handler RoundRobin.

4.6 Case Study: Cooperative Multitasking

Like generators and coroutines, cooperative multitasking and async/await can be implemented as a library [Dolan et al. 2017; Leijen 2017b]. Programs can use the Coop effect to fork and suspend processes. A process is an effectful program Eff<Void, Void>. The RoundRobin handler[11] in Figure 8 implements a scheduler that keeps a queue of all running processes in its handler state. Forking a process *p* is implemented by using *flip* to get a boolean and then either starting *p* and discarding the continuation when *p* returns or continuing as normal otherwise.

Yielding is implemented by enqueueing the continuation of the process and immediately returning. The handler will then pick the next process to execute. A program that uses Coop is executed by first handling the effect using RoundRobin and then running the scheduler with *run*.

5 IMPLEMENTATION OF THE TYPE SELECTIVE CPS TRANSFORMATION

Section 3.1 illustrated the type-selective CPS translation by example as a source-to-source transformation. This section presents the implementation of the translation on the level of bytecode and shows how we handle control flow elements like jumps and exceptions. The translation is interesting in that it uses Java 8 closures to create continuations. This is in contrast to other bytecode translations which we will compare in Section 6.1.

We implemented the transformation described in this section using the OPAL framework[12] by Eichberg and Hermann [2014] for static analysis and synthesis of JVM bytecode. Our implementation of the CPS transformation as well as all other components of Effekt can be found online[13].

We minimize the presentation to a relevant core set of language features, leaving out details that distract from the essence of the transformation: Storing all necessary function local state in closures and inserting calls to the Effekt API as described in Section 3. We model the JVM and thus assume an abstract machine with registers (also referred to as *locals*, since they are function local), an operand stack (also referred to as *operands*, again function local) and a frame stack (commonly referred to as *stack*). We use the term *function local state* to refer to the values stored in locals and operands at a given time in the execution of a method. We also assume the JVM calling convention that function arguments are pushed on the operand stack by the caller, but accessible as locals by the callee, starting from register index 0. We assume every bytecode instruction is labeled, but omit labels that we never refer to.

We explain the transformation on an effectful example method *doLoop*:

```
boolean doLoop () throws Effects {
  Reader r = Readers.getReader (); // static method
  loop: try {while ('\n' ≠ r.read ()) { } }
        catch (MyExc e) {return false; }
  exit: return true;
}
```

The bytecode of the method *doLoop* is shown in Figure 9a. The method *doLoop* will perform effect calls to *r.read* () until the result is either a newline or *r.read* () raises a native exception. The example includes exception handling to illustrate in more detail how the bytecode instrumentation interacts with native exceptions. As in the JVM, exception-handling is modeled external to the list of bytecode instructions of a method and exception handlers are given in the form of regions as

excregion *tryStartLabel tryEndLabel catchLabel exception*

[11]The type parameters of Handler are Void, they are omitted due to lack of space. *pure* just returns **null**.
[12]http://www.opal-project.de/
[13]http://github.com/b-studios/jvm-effekt

In our example, let us assume an exception region **excregion** *loop break catch MyExc* is in place. That is, if an exception of type *MyExc* is raised in the dynamic region between *loop* and *break* execution will be continued at *catch*.

```
method doLoop 1 throws Effects {
          invoke Readers.getReader 0
          store 1
  loop:   const '\n'
          load 1 // load Reader from local 1
  op:     invoke Reader.read 1
          ifeq exit
  break:  goto loop
  catch:  const false
          return
  exit:   const true
          return
}
excregion loop break catch MyExc
```

(a) Bytecode of method *doLoop*.

```
S⟦doLoop⟧      = method doLoop 1 throws Effects {
                    load 0        // load reference to 'this'
                    closure Frame.enter doLoop_init 1
                    const false // load dummy value
                    return
                  }
E⟦doLoop⟧_init = method doLoop_init 1 {
                    goto entry_init
                    ...
                  }
E⟦doLoop⟧_op   = method doLoop_op 2 {
                    goto entry_op
                    ...
                  }
```

(b) Generated methods – method bodies in Fig. 9d.

$effCalls_{doLoop} = [init, op]$ $operands_{init} = 0$ $operands_{op} = 1$ $tmpLocal_{doLoop} = 2$ $locals_{init} = [0]$ $locals_{op} = [1]$

(c) Meta information as obtained by static analysis.

```
entry_init: invoke Readers.getReader 0
            store 1
loop:       const '\n'
            load 1
op:         store 2                          // save call operands
            load 1                           // load live locals
            closure Frame.enter doLoop_op 2  // close over two values
            invoke Effekt.push 1             // push closure to stack
            load 2                           // restore call operands
            invoke Reader.read 1
            return_void
↩
```

```
entry_op: load 0                           // load arguments
          load 1
          store 1                          // restore locals
          invoke Effekt.result 0           // get result
          ifeq exit
break:    goto loop
catch:    const false
          invoke Effekt.returnWith 1       // store result
          return_void
exit:     const true
          invoke Effekt.returnWith 1       // store result
          return_void
```

(d) Result of translating the instructions of method *doLoop*.

Fig. 9. Example of translating a method *doLoop*.

The syntax of the term-language is summarized in Figure 10a. For simplicity of the presentation, we do not concern ourselves with types and thus choose a uni-typed term-language. The translation does not distinguish instance methods and static methods. Hence, we only include a single method definition that consists of a name, a list of exception handler regions (the *exception table*), a list of potentially raised exceptions and a list of labeled instructions. The syntax of bytecode instructions *Instr* in Figure 10a includes instructions to load constant values (**const** *v*) to the operand stack, instructions to load from and store into function local registers (**load** *index*, **store** *index*) and control flow instructions (**return**, **throw**, **ifeq**[14] and **goto**). Finally, as with method declarations,

[14]For our example, we assume **ifeq** *label* pops two values and jumps to the given *label* if the two values are equal.

we only include a single form of method invocation **invoke** *name arity* that subsumes static and virtual method calls. Virtual method calls take the receiver as the first argument and thus always have an arity greater than or equal to one.

$$
\begin{array}{lcll}
\mathit{instr} & \in & \mathit{Instr} ::= & \textbf{const}\ \mathit{Value} \mid \textbf{load}\ \mathbb{N} \mid \textbf{store}\ \mathbb{N} \\
 & & & \mid \textbf{return} \quad \mid \textbf{throw} \\
 & & & \mid \textbf{goto}\ \mathbb{L} \quad \mid \textbf{ifeq}\ \mathbb{L} \\
 & & & \mid \textbf{invoke}\ \mathit{Name}\ \mathbb{N} \\
 & & & \mid \textbf{closure}\ \mathit{Name}\ \mathit{Name}\ \mathbb{N}
\end{array}
$$

$$
\begin{array}{lcl}
m & \in & \mathit{Method} ::= \textbf{method}\ \mathit{Name}\ \mathbb{N}\ \textbf{throws}\ \overline{\mathit{Name}}\ \{\overline{\mathbb{L}\colon \mathit{Instr}}\} \\
\mathit{label} & \in & \mathbb{L} \\
\mathit{name} & \in & \mathit{Name}
\end{array}
$$

(a) Syntax of methods and bytecode instructions.

(T-Stub) $\boxed{\mathcal{S}[\![\cdot]\!]\colon \mathit{Method} \to \mathit{Method}}$

$$
\begin{array}{l}
\mathcal{S}[\![\ \textbf{method}\ \mathit{name}\ \mathit{arity}\ \textbf{throws}\ \overline{\mathit{exc}}\ \{\ \overline{\mathit{instr}}\ \}]\!] = \\
\quad \textbf{method}\ \mathit{name}\ \mathit{arity}\ \textbf{throws}\ \overline{\mathit{exc}}\ \{ \\
\qquad \mathit{saveState}\ (\mathit{init}) \\
\qquad \mathit{loadDummyResult} \\
\qquad \textbf{return} \\
\quad \}
\end{array}
$$

(T-Entrypoint) $\boxed{\mathcal{E}[\![\cdot]\!]_{\mathit{eff}}\colon \mathit{Method}\ \mathbb{L} \to \mathit{Method}}$

$$
\begin{array}{l}
\mathcal{E}[\![\ \textbf{method}\ \mathit{name}\ \mathit{arity}\ \textbf{throws}\ \overline{\mathit{exc}}\ \{\ \overline{\mathit{instr}}\ \}]\!]_{\mathit{eff}} = \\
\quad \textbf{method}\ \mathit{name}_{\mathit{eff}}\ \mathit{closureArity}_{\mathit{eff}}\ \textbf{throws}\ \emptyset\ \{ \\
\qquad \textbf{goto}\ \mathit{entry}_{\mathit{eff}} \\
\qquad \overline{\mathcal{I}[\![\mathit{instr}]\!]} \\
\quad \}
\end{array}
$$

(b) Transformation of effectful methods

(T-Invoke-Eff) $\boxed{\mathcal{I}[\![\cdot]\!]\colon (\mathbb{L}\colon \mathit{Instr}) \to \overline{\mathbb{L}\colon \mathit{Instr}}}$

$$
\begin{array}{rl}
\mathcal{I}[\![\ \mathit{eff}\colon & \textbf{invoke}\ \mathit{fun}\ \mathit{arity}]\!]\ \ \textbf{if}\ \ \mathit{effectful}\ (\mathit{fun}) = \\
\mathit{eff}\colon & \mathit{saveCallOperands}\ (\mathit{tmpLocal}_m,\ \mathit{arity}) \\
 & \mathit{saveState}\ (\mathit{eff}) \\
 & \mathit{restoreCallOperands}\ (\mathit{tmpLocal}_m,\ \mathit{arity}) \\
 & \textbf{invoke}\ \mathit{fun}\ \mathit{arity} \\
 & \textbf{return}_{\textbf{void}} \\
\mathit{entry}_{\mathit{eff}}\colon & \mathit{restoreState}\ (\mathit{eff}) \\
 & \textbf{invoke}\ \text{Effekt}.\mathit{result}\ 0
\end{array}
$$

(T-Return)

$$
\begin{array}{rl}
\mathcal{I}[\![\ \mathit{label}\colon & \textbf{return}]\!] = \\
\mathit{label}\colon & \textbf{invoke}\ \text{Effekt}.\mathit{returnWith}\ 1 \\
 & \textbf{return}_{\textbf{void}}
\end{array}
$$

(T-Other)

$$
\mathcal{I}[\![\ \mathit{label}\colon\ \ \mathit{instr}]\!] = \mathit{label}\colon \mathit{instr}
$$

(c) Transformation of bytecode instructions

$$
\begin{array}{l}
\mathit{saveCallOperands}\colon\ \ \mathbb{N} \times \mathbb{N} \to \overline{\mathbb{L}\colon \mathit{Instr}} \\
\mathit{saveCallOperands}\ (\mathit{first},\ n) = \\
\qquad \mathit{storeLocals}\ (\mathit{first}\ \mathit{to}\ (\mathit{first} + n - 1))
\end{array}
$$

$$
\begin{array}{l}
\mathit{restoreCallOperands}\colon \mathbb{N} \times \mathbb{N} \to \overline{\mathbb{L}\colon \mathit{Instr}} \\
\mathit{restoreCallOperands}\ (\mathit{first},\ n) = \\
\qquad \mathit{loadLocals}\ ((\mathit{first} + n - 1)\ \mathit{downTo}\ \mathit{first})
\end{array}
$$

$$
\begin{array}{l}
\mathit{saveState}\colon\ \ \mathbb{E} \to \overline{\mathbb{L}\colon \mathit{Instr}} \\
\mathit{saveState}\ (\mathit{eff}) = \\
\qquad \mathit{loadLocals}\ (\mathit{locals}_{\mathit{eff}}) \\
\qquad \textbf{closure}\ \text{Frame}.\mathit{enter}\ \mathit{name}_{\mathit{eff}}\ \mathit{closureArity}_{\mathit{eff}} \\
\qquad \textbf{invoke}\ \text{Effekt}.\mathit{push}\ 1
\end{array}
$$

$$
\begin{array}{l}
\mathit{restoreState}\colon \mathbb{E} \to \overline{\mathbb{L}\colon \mathit{Instr}} \\
\mathit{restoreState}\ (\mathit{eff}) = \\
\qquad \mathit{loadLocals}\ (0\ \mathit{to}\ (\mathit{closureArity}_{\mathit{eff}} - 1)) \\
\qquad \mathit{storeLocals}\ (\mathit{reverse}\ (\mathit{locals}_{\mathit{eff}}))
\end{array}
$$

(d) Helper functions

Fig. 10. Type selective CPS translation via bytecode transformation.

5.1 Translation of Methods

Only effectful methods that are marked as throwing Effects exceptions are translated, all other methods of a class are copied unchanged.

As was seen in the source-to-source transformation in Figure 4, for one effectful method m, we generate multiple methods: a single method stub, using the translation function $\mathcal{S}[\![\cdot]\!]$ and one entrypoint method for each effect call at label eff using the translation function $\mathcal{E}[\![\cdot]\!]_{\mathit{eff}}$ (both translation functions are defined in Figure 10b). Every effect call $\mathit{eff}: \textbf{invoke}\ \mathit{fun}\ \mathit{arity}$ inside a given method m gives rise to an entrypoint uniquely identified by the label eff. The entrypoint itself is labeled $\mathit{entry}_{\mathit{eff}}$ and represents the continuation of the method m after the effect call returned. For consistency, we also treat the initial entrypoint at label init as effect call. The special entrypoint $\mathit{entry}_{\mathit{init}}$ refers to the label of the first original instruction of the method. By explicitly pushing the initial entrypoint, we perform trampolining for each effect call. For the method doLoop, we thus generate three methods: the method stub doLoop and the two entrypoint methods $\mathit{doLoop}_{\mathit{init}}$ and $\mathit{doLoop}_{\mathit{op}}$ (Figure 9b).

Rule T-Stub generates the method stub that first saves the local state and then immediately returns a dummy value. At that point, the local state only consists of the arguments supplied to the function call. As we will see shortly, $\mathit{saveState}$ thus pushes a closure that closes over the call arguments and resumes with method $\mathit{doLoop}_{\mathit{init}}$ when invoked. The returned result of the stub method will never be used, hence $\mathit{loadDummyResult}$ can load any constant value.

Rule T-Entrypoint generates two static entrypoint methods $\mathit{doLoop}_{\mathit{init}}$ and $\mathit{doLoop}_{\mathit{op}}$. The bodies of the two methods are exactly the same after the initial **goto** instruction and are given in Figure 9d. The only difference is the initial jump. Since generating almost identical methods for each entrypoint leads to unnecessary growth of the class file, in our implementation of Effekt, we perform dead code elimination after generating the bytecode. Using closures to save state, only saving live variables and performing dead code elimination ultimately results in code which is very close to handwritten code in continuation passing style (as in Figure 4c).

5.2 Saving Function Local State

To generate state saving and restoring code, we use the following information about a method m, which we obtain by static analysis:

- the set of all labels corresponding to effect calls, also including the first label $\mathit{entry}_{\mathit{init}}$ but no effect tail calls – $\mathit{effCalls}\,(m) \in \overline{\mathbb{L}}$,
- the index of the first free local register, not used by the original instructions of method m – $\mathit{tmpLocal}_m \in \mathbb{N}$.

Likewise, for each effect call eff in a method m, the transformation uses the following information which we again obtain by static analysis:

- the number of operands on the operand stack after the effect call (not including the result of the effect call) – $\mathit{operands}_{\mathit{eff}} \in \mathbb{N}$,
- the list of indices of local registers which are alive after the effect call – $\mathit{locals}_{\mathit{eff}} \in \overline{\mathbb{N}}$,

For method doLoop, the static information is given in Figure 9c. We also define $\mathit{closureArity}_{\mathit{eff}}$ to equal $\mathit{operands}_{\mathit{eff}} + |\ \mathit{locals}_{\mathit{eff}}\ |$, referring to the total number of values that need to be stored in the closure, that is all operands and the number of locals which are live *after* the effect call. To actually save the state, the meta function $\mathit{saveState}$ generates code that stores those parts of the function local state which are necessary to resume the execution of the function. This includes all operands (after the effect call) and the contents of all registers which correspond to live local variables. This is achieved in three steps:

(1) all live local variables are loaded to the operand stack; the operands do not need to be loaded since they already are on the operand stack
(2) a new instance of a Frame is created as a lambda using the given method $\mathit{name}_{\mathit{eff}}$ as the body of the lambda and closing over $\mathit{closureArity}_{\mathit{eff}}$-many values on the operand stack;

(3) finally, the newly created frame is pushed using Effekt.*push*.

In JVM bytecode, closures are created by issuing a specific **invokedynamic** call to a *lambda metafactory*. We refer to this call only in its specialized form as

closure *interfaceName name arity*

The call to **closure** is provided with a $name \in Name$ of a method which serves as the implementation of the lambda, implementing a single-abstract-method interface $interfaceName \in Name$[15] and an $arity \in \mathbb{N}$ which specifies the number of values the lambda should close over. The JVM runtime passes the closed-over values as additional arguments to the implementing method when the closure is applied.

5.3 Translation of Instructions

To generate the bodies of entrypoint methods, Figure 10c defines the semantic function $\mathcal{I}[\![\cdot]\!]$ which specifies how labeled instructions are translated. The result of translating the body of *doLoop* can be found in Figure 9d. Figure 10d gives the implementation of some of the helper functions. They are meta functions and thus expand at translation time to generated code. Given a list of register indices, the unlisted functions *loadLocals* and *storeLocals* generate bytecode that loads from (respectively stores to) all given locals. We use the notation x *to* y to denote a range of indexes from x to y, both ends inclusive. Similarly, x *downTo* y denotes a decreasing range.

The translation of bytecode instructions behaves as identity (Rule T-Other) except for effect calls (Rule T-Invoke-Eff) and returns (Rule T-Return). To stress, non-effectful calls don't require any modification. To translate effect calls, rule T-Invoke-Eff saves the function local state, performs the effect call and then suspends the method by returning to the trampoline ($\mathbf{return}_{\mathbf{void}}$). In the translated program, all jumps to the effect call within m should point to the instrumented call instead. Therefore we change the label *eff* to point to the first instruction of the state saving code. This automatically also affects exception regions that mention *eff*. As in our *doLoop* example, the effect call might require arguments that reside on the operand stack at the time of state-saving. To account for this, we use temporary locals (which will not be stored in the closure) to set the call operands temporarily aside.

The remainder of the function after the effect call is labeled with $entry_{eff}$. Since it immediately follows a return, this part of the code is only reachable by the **goto** $entry_{eff}$ in the corresponding entrypoint method. At that time, all function state necessary for resumption has been passed as arguments and is thus stored in first $closureArity_{eff}$-many registers. For our example of *doLoop*, the code at label $entry_{op}$ assumes that one operand (the constant `'\n'`) and one local (an instance of type Reader) have been passed as arguments and are thus available via registers 0 and 1. Before the function can be resumed, its state needs to be reset to where it has been left off. The meta function *restoreState* loads all saved operands and locals (in this order) to the operand stack. It then writes the locals to the correct registers (in reverse order). The result of the previous effect call is obtained by Effekt.*result*. If the previous effect call exited abnormally by throwing an exception, Effekt.*result* as implemented in Figure 5, will re-raise this exception. Since we already restored all operands and locals, the exception will be raised in the correct context and trigger the correct exception handlers. Being defined in terms of labels, our translation does not need to modify the exception table. The rule T-Return replaces every return with a call to Effekt.*returnWith* to install the second half of the special calling convention.

[15]For the translation, *interfaceName* will always be Frame.*enter*

6 DISCUSSION AND RELATED WORK

We discuss design decisions, related work, performance and future work. Existing implementations of libraries and languages for (algebraic) effect handlers are either translations to a high level language or involve a custom runtime implementation. High level implementations translate effect handlers into delimited continuations [Brachthäuser and Schuster 2017; Kammar et al. 2013; Kiselyov and Sivaramakrishnan 2016], free monads [Kiselyov and Ishii 2015] or perform a source to source CPS translation into another high-level language [Hillerström et al. 2017; Leijen 2017c]. Other implementations require a custom runtime that supports stack manipulation [Bauer and Pretnar 2015; Dolan et al. 2017] or setjump / longjump [Leijen 2017a]. In this paper, we explore a new implementation technique for effect handlers in terms of a CPS transformation of bytecode. The discussion is split accordingly into a discussion of the bytecode transformation and a discussion of the Effekt framework for programming with effect handlers as a whole.

6.1 Continuations on the Java Virtual Machine

We review related work on (delimited) continuations and CPS transformations in the context of the Java virtual machine. While implementations that modify the JVM exist [Dragos et al. 2007; Stadler et al. 2009] or are under development [Pressler 2017]. While those specialized runtimes could potentially be used as backend for our effect handler library, here we will focus on library solutions. We compare our CPS transformation with three other Java projects that perform bytecode instrumentation. A library for fibers "Quasar"[16], a library for one-shot continuations "JavaFlow"[17] and a library for coroutines[18].

Continuation instantiation. Approaches to capture the continuation can be characterized by the point in time the continuation is constructed. CPS transformations create the continuation *before* the execution of an effectful call. The continuation is thus always immediately available. This is how we implemented Effekt. It is also the case for implementations of effect handlers that rely on CPS [Hillerström et al. 2017] and corresponding monadic implementations in eager languages like Scala Effekt [Brachthäuser and Schuster 2017]. Quasar also explicitly stores all function state before entering an effect call. Another approach is to instantiate the continuation only when it is needed. Typically, to signal that the continuation needs to be captured, the effectful function can use a special exception [Loitsch 2007; Pettyjohn et al. 2005; Sekiguchi et al. 2001], sum types [Kiselyov and Sivaramakrishnan 2016] or global flags (JavaFlow, Coroutines).

Stack Restoration. Some implementations are designed to fully restore JVM stack when a continuation is resumed (Quasar, JavaFlow). This simplifies integration with exceptions, stack traces and debuggers. Full restoration of the stack is a technical consequence of not having a first class representation of continuation frames. In consequence, all bytecode continuation libraries in Java that we are aware of resume a continuation by replaying all function calls. However, restoring the stack is always linear in the depth of the stack since all function calls need to be replayed.[19] In contrast to that, in Effekt we explicitly reify each continuation frame as a closure and upon resumption we enter the first frame without restoring the Java stack. While this helps to reduce the asymptotic complexity from quadratic to linear, stack traces in Effekt only show very few frames which can impede debugging.

[16] http://docs.paralleluniverse.co/quasar
[17] http://commons.apache.org/sandbox/commons-javaflow/
[18] https://github.com/offbynull/coroutines
[19] For Quasar, this observation has also been made by Aleksandar Prokopec (Oracle Labs) – private communication.

Function state representation. The state necessary to resume a suspended function consists of the function local state and an entrypoint label. It can be represented and stored in different ways. The entrypoint label can be encoded as a number that will be dispatched upon with a switch statement at the beginning of the method. This is commonly combined with storage of the function local data in a stack like data structure (Quasar, JavaFlow, Sekiguchi et al. [2001], Bierman et al. [2012]). An alternative is to replace the switch by dynamic dispatch and to store the function data in a closure [Pettyjohn et al. 2005]. This is how Effekt is implemented.

Multiple resumptions. We designed Effekt in a way that continuations can naturally be resumed multiple times. In implementations supporting only *one-shot* continuations, state update can be destructive, which makes it easier to implement continuations efficiently [Dolan et al. 2015]. While Effekt maintains one global immutable runtime stack (as in Figure Figure 5), Quasar and JavaFlow maintain one mutable stack per delimited continuation / fiber. In such a setting, multiple resumptions are implemented by deeply cloning the corresponding stack and all nested stacks before resuming. In Effekt function local state is copied into immutable frames. Also stack segments are immutable and can be shared across multiple resumptions.

6.2 Relation to other (Algebraic) Effect Handler Libraries and Languages

Most implementations of libraries and languages for (algebraic) effects are based on a *deep embedding* of effect operations. They reify effect operations as alternatives in a sum type. For instance, the *flip* effect operation would be reified as a constructor of an algebraic data type Amb. Handlers then use pattern matching to interpret the reified effect operations [Bauer and Pretnar 2015; Hillerström et al. 2017; Kiselyov and Ishii 2015; Kiselyov and Sivaramakrishnan 2016; Leijen 2014]. To mix programs with different effects means to extend an open union type of reified effect operations.

In contrast, Effekt builds on a *shallow embedding* [Carette et al. 2007; Hudak 1998] of effect operations. Shallow embeddings can be structured in a pleasingly extensible way [Oliveira and Cook 2012]. Interpretation of effect operations is moved from (external) pattern matching to (internal) dynamic dispatch which makes a shallow embedding of effect operations a good fit for object oriented programming languages. Kammar et al. [2013] base their library implementation of algebraic effect handlers on Haskell type classes, effectively performing a shallow embedding. Using type classes helps Kammar et al. to achieve good performance results since it prevents the materialization of constructors for effect operations. Brachthäuser and Schuster [2017] present a monadic library implementation of effect handlers for Scala using implicit parameters and handler passing. They show how programming with algebraic effects is an instance of the expression problem and explore how shallow embedded handlers open up different dimensions of extensiblity.

One advantage of the shallow embedding is that it simplifies typing. We use dynamic dispatch instead of implementing a pattern matching interpreter. This helps us to avoid advanced typing features, such as type constructor polymorphism [Kiselyov et al. 2013] or generalized algebraic data types [Kiselyov and Ishii 2015].

Combining OO with effect handlers, we define `use` as a method on `Handler`. As a method, it naturally shares the type of the effect domain with its implementing class. Others require path dependent types for the same purpose [Brachthäuser and Schuster 2017]. In contrast, Effekt as presented in this paper is designed to remove requirements on the type system and to blend in with Java programming paradigms. The bytecode instrumentation and the `Stateful` interface both enable the direct use of mutable state and effect signatures are modeled by simple interfaces (as compared to traits with type members and path dependent types in Scala Effekt).

Leijen [2017a] explicitly tags each effect operation in a handler with information about how the continuation is used to implement important optimizations. Similarly explicit, in Effekt, handlers

Table 1. Performance of bytecode instrumentation implementations. Runtime in ms, lower is better.

	Time in ms (Confidence Interval)					
Benchmark	**Baseline**	**Effekt**	**Effekt$_{\text{opt}}$**	**Coroutines**	**Quasar**	**JavaFlow**
Stateloop 1M	1.61 ±0.09	29.76 ±2.57	1.91 ±0.04	5.52 ±0.35	69.02 ±2.59	14.82 ±0.48
RecursiveOnce 1K	0.01 ±0.0	0.69 ±0.22	0.34 ±0.01	0.07 ±0.0	0.23 ±0.03	8.18 ±0.19
RecursiveMany 1K	0.01 ±0.0	1.05 ±0.38	0.4 ±0.07	10.29 ±1.41	68.07 ±2.07	3363.74 ±23.46
Skynet 1M	2.74 ±0.03	171.34 ±5.55	62.13 ±3.87	35.19 ±2.51	762.1 ±155.95	1277.51 ±54.18
SkynetSuspend 1M	2.74 ±0.03	414.56 ±9.2	147.4 ±5.44	50.46 ±2.95	1113.15 ±112.78	7198.72 ±122.56

capture the continuation with `use`. *Tail resumptive* handlers don't need to capture the continuation and don't call `use`.

In other libraries and languages for effect handlers, an effect operation implicitly resolves to the dynamically closest handler implementation. In contrast, we require the user to explicitly select the handler to use. This has the advantage that no confusion arises when multiple handlers for the same effect are present and that we avoid any search for the correct handler implementation in some kind of handler stack. Explicitly selecting which handler to use also avoids the problem of *effect encapsulation* [Biernacki et al. 2017; Lindley 2018], that is, handlers accidentally handle effects. The disadvantage is that explicit handler selection is more verbose and fragile to changes.

Due to our design decision of a shallow embedding of effect handlers, handlers in Effekt are *deep handlers* [Kammar et al. 2013]. That is, all effect operations in the continuation captured by `use` will automatically be handled recursively by the very same handler. However, if a *shallow handler* semantics is required, it can be achieved by reifying the command-response trees of selected effect operations. Similar to the conversion from shallow to deep embedding, a reifying effect handler interprets a program of type *R* into a free-structure *Free<R>* [Kiselyov et al. 2013] which then can be interpreted step-by-step. While this encoding is possible, it can lead to performance problems and memory leaks.

6.3 Performance

We report on some preliminary performance results. Like the discussion, the evaluation of performance is split into two parts: A part on the CPS transformation and a part on Effekt as a library for programming with effect handlers. All benchmarks were executed on a 2.5 GHz Intel Core i7 with 16GB of memory using the ScalaMeter, a state of the art JVM benchmarking library.

We also show the performance results for a variant of Effekt that implements several optimizations: Continuation frames are not materialized upfront, but only when needed. To avoid push-pop-enter cycles, the initial entrypoint is not explicitly pushed but inlined. Only methods that contain at least one non-tail effect call are instrumented. We refer to this variant as Effekt$_{\text{opt}}$. The user programs using Effekt don't need to be changed.

Performance of the Bytecode Instrumentation. We evaluate the performance of our CPS transformation comparing with Quasar 0.7.9, a recently maintained fork of JavaFlow[20] in version 2.2.1 and Java Coroutines in version 1.4.0. We also measure the overhead compared to a baseline that does not capture continuations. All libraries perform some sort of bytecode transformation to support capturing the continuation. Since each of the libraries targets a particular domain (coroutines / fibers), capturing the continuation also involves additional overhead specific to the target application. Where possible, we reduced this overhead by disabling features – focusing on the

[20]https://github.com/vsilaev/tascalate-javaflow

Table 2. Performance of effect libraries. Runtime in ms, lower is better.

	Time in ms (Confidence Interval)			
Benchmark	**Effekt**	**Effekt$_{\text{opt}}$**	**Scala Effekt**	**Scala Eff**
Countdown 10K	3.35 ±0.07	2.47 ±0.12	6.07 ±0.32	34.39 ±2.59
Countdown8 1K	1.31 ±0.39	1.77 ±0.1	2.31 ±0.12	36.92 ±3.0
NQueens (10)	19.5 ±0.38	16.09 ±0.19	40.95 ±0.54	49.89 ±2.17

continuation capturing aspect, only. The results of the measurements can be found in Table 1. To assess the instrumentation overhead, the Stateloop benchmark counts down from one million to zero, performing some computation work at each step but not capturing the continuation. We can see that most of the overhead of creating continuation frames is eliminated in the alternative Effekt$_{\text{opt}}$. Java Coroutines save the function local state in arrays before entering a potentially suspending function call. This is unnecessary for the Stateloop benchmark, which does not suspend. To measure performance of capturing the continuation, the RecursiveOnce and RecursiveMany benchmarks also count down, but as recursive functions. For the first benchmark we suspend the computation once before returning the result (at stack-depth 1,000); correspondingly, for the second one we suspend once at every recursive call. Resuming continuations is linear in stack depth for all implementations but the Effekt implementations. In consequence, for the other implementations, RecursiveMany has a running time that is quadratic in N while it is linear for Effekt and Effekt$_{\text{opt}}$. To measure performance of delimited continuations, the Skynet benchmark[21] recursively spawns ten fibers until one million are created, performs some computation and aggregates the results. Each fiber corresponds to one delimited continuation. The Skynet variant never suspends, but just creates the fibers which immediately return. The SkynetSuspend variant in contrast suspends each fiber once before returning, resulting in one million continuations to be captured and resumed. Quasar and JavaFlow maintain one stack per delimited continuation / fiber each pre-allocating memory to store the function state. JavaFlow additionally maintains one stack per primitive type and copies the stack on every resumption. This leads to several million arrays copies. The Coroutines library is optimized for one shot continuations and large parts of the library are inlined in the generated bytecode. It also does not suffer from the linear stack restoration in the Skynet benchmark since the stack size of each fiber on suspension is at most one.

Performance of the Effect Library. To evaluate the performance of the overall framework, we compare Effekt with two other effect libraries. A monadic implementation of effect handlers in Scala "Scala Effekt"[22] and the effect library "Scala Eff"[23] which is based on freer monads [Kiselyov and Ishii 2015]. The results of the measurements can be found in Table 2. The CountDown8 benchmark layers eight state effects over one ambiguity effect and flips once before returning. NQueens is an effect library benchmark from the literature [Kammar et al. 2013]. The benchmarks show improvements of 2x compared to Scala Effekt and 2.5-28x compared to Scala Eff. We account the biggest performance improvement to optimizing tail resumptive operations to just be dynamic method calls. Other performance improvements compared to Scala Effekt are inlining the monadic Scala code by bytecode instrumentation and only capturing the continuation on demand (Effekt$_{\text{opt}}$).

[21] https://github.com/atemerev/skynet
[22] http://b-studios.de/scala-effekt/
[23] http://atnos-org.github.io/eff/

7 CONCLUSIONS

We presented the first library design for programming with effect handlers in Java. We showed how such a library can be implemented in terms of a CPS transformation and multi-prompt delimited continuations. Our CPS transformation allows trampolining, multiple resumptions and is competitive in its performance.

ACKNOWLEDGMENTS

We would like to thank Aleksandar Prokopec and Daan Leijen for inspiring discussions that helped us to improve the design and implementation of EFFEKT. We are also very grateful for the valuable feedback by Sam Lindley and the anonymous reviewers. This work was supported by DFG project OS 293/3-1.

REFERENCES

Andrej Bauer and Matija Pretnar. 2013. An effect system for algebraic effects and handlers. In *International Conference on Algebra and Coalgebra in Computer Science*. Springer, 1–16.

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

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

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM.

Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: Extensible Algebraic Effects in Scala (Short Paper). In *Proceedings of the International Symposium on Scala*. ACM.

Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the International Conference on Functional Programming*. ACM, 133–144.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer LNCS 4807, 222–238.

Melvin E Conway. 1963. Design of a separable transition-diagram compiler. *Commun. ACM* 6, 7 (1963).

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*.

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

Iulian Dragos, Antonio Cunei, and Jan Vitek. 2007. Continuations in the Java virtual machine. In *Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2007)*. Technische Universität Berlin.

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

Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*. ACM.

Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. 1999. Trampolined Style. In *Proceedings of the International Conference on Functional Programming*. ACM, 18–27.

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

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

Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *Proceedings of the Conference on Software Reuse*. IEEE Computer Society Press, 134–142.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the International Conference on Functional Programming*. ACM, 145–158.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the Haskell Symposium*. ACM, 94–105.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the Haskell Symposium*. ACM, 59–70.

Oleg Kiselyov and KC Sivaramakrishnan. 2016. Eff directly in OCaml. In *ML Workshop*.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*.

Daan Leijen. 2016. *Algebraic Effects for Functional Programming.* Technical Report. MSR-TR-2016-29. Microsoft Research technical report.

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*. ACM, 16–29.

Daan Leijen. 2017c. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. 486–499.

Sam Lindley. 2018. Encapsulating effects, In Algebraic Effect Handlers go Mainstream (Dagstuhl Seminar 18172). *Dagstuhl Reports* 8, 4.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 500–514.

Florian Loitsch. 2007. Exceptional continuations in JavaScript. In *Workshop on Scheme and Functional Programming*.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer LNCS 7313, 2–27.

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. 2005. Continuations from Generalized Stack Inspection. In *Proceedings of the International Conference on Functional Programming*. ACM, 216–227.

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

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

Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 217–232.

Ron Pressler. 2017. *Loom Project: Fibers and Continuations for the Java Virtual Machine.* OpenJDK Project. HotSpot Group. http://mail.openjdk.java.net/pipermail/discuss/2017-September/004390.html

Aleksandar Prokopec and Fengyun Liu. 2018. Theory and practice of coroutines with snapshots. In *Proceedings of the European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM annual conference*. ACM, 717–740.

Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. 2001. Advances in Exception Handling Techniques. Springer-Verlag, Heidelberg, Berlin, Germany, Chapter Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling, 217–233.

Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. 2009. Lazy continuations for Java virtual machines. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*. ACM, 143–152.

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