JEff: Objects for Effect

Pablo Inostroza
Centrum Wiskunde & Informatica (CWI)
The Netherlands
pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
University of Groningen, Groningen
The Netherlands
storm@cwi.nl

Abstract

Effect handling is a way to structure and scope side-effects which is gaining popularity as an alternative to monads in purely functional programming languages. Languages with support for effect handling allow the programmer to define idioms for state, exception handling, asynchrony, backtracking, etc. from within the language. Functional programming languages, however, prohibit certain patterns of modularity well-known from object-oriented languages. In this paper we introduce JEff, an object-oriented programming language with native support for effect handling, to provide first answers to the question what it would mean to integrate object-oriented programming with effect handling. We illustrate how user-defined effects could benefit from interface polymorphism, and present its runtime semantics and type system.

CCS Concepts • **Software and its engineering → Object oriented languages**; **Control structures**;

Keywords object-oriented languages, effect handling

ACM Reference Format:
Pablo Inostroza and Tijs van der Storm. 2018. JEff: Objects for Effect. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3276954.3276955

1 Introduction

Effect handlers [17, 18] are a programming language mechanism to structure, scope, and compose side-effects in purely functional languages. Languages that support effect handling natively, such as Koka [14], Eff [4], Frank [15], allow side effects (state, IO, exceptions, coroutines, asynchrony, etc.) to be defined within the programming language, as libraries. Effect handlers have been touted as a simpler and more composable way of programming with effects than other techniques, such as, for instance, monads. As a result, perhaps, existing research on effect handling has mainly focused on functional programming languages.

Functional programming languages, however, prohibit certain patterns of modularity and reuse well-known in object-oriented languages. Yet the huge success of object-orientation in practice [1] seems to suggest these mechanisms are valuable programming tools. This raises the question: what would it mean to integrate effect handling with object-oriented programming?

In this paper we present first steps towards answering this question in the form of JEff, an object-oriented Java-like language without side-effects or inheritance, but with built-in support for programming with effect handlers. As a simple example, consider the code snippet shown in Figure 1. It defines an effect interface `StdOut` declaring a single effect method `print` for printing to the console. The class `MyStdOut` implements the interface and also marks itself as being a `Handler`; we defer the details of implementing handlers to Section 2. The `StdOut` interface is used in the `hello` method, which declares it as a required effect using `@`. In its body it calls the `print` method. Finally, the `main` method – which is pure – installs a `MyStdOut` handler with the **`with`**-construct, providing the printing capability to `hello`. Since `MyStdOut` models the console as a list of strings, this, together with the unit value, will be the result of `main`.

The example already highlights the most important aspect of JEff, namely that the declaration of an effect is decoupled

```
interface StdOut { eff Unit print(String s) }

class MyStdOut<T>(List<String> o) implements
  StdOut, Handler<Tuple<List<String>, T>, T> {...}

class Main() {
  Unit hello()@StdOut = StdOut::print("Hello world!")

  Tuple<List<String>, Unit> main() =
    with (new MyStdOut<Unit>([])) { this.hello() }
}
```

Figure 1. Skeleton of a simple JEff program

Onward! '18, November 7–8, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6031-9/18/11…$15.00
https://doi.org/10.1145/3276954.3276955

from its implementation by a handler through interfaces. For instance, the `StdOut` interface can be implemented by multiple handler classes like `MyStdOut`, but encapsulating different internal representations. Nevertheless, the `hello` method only refers to the effect interface, and hence can be executed over any such implementation. JEff thus leverages dynamic dispatch and interface-based encapsulation (two of the corner-stones of object-oriented programming) for defining effects.

The contributions of this paper can be summarized as follows:

- We present JEff, the first object-oriented language with native support for effect handling (Section 2).
- We illustrate how common effects like exception handling (Section 2.2) and state (Section 2.3) are realized in JEff, thus providing an object-oriented introduction to effect handling, and how object-oriented effect handling facilitates structuring interpreters and ad hoc overloading of effects (Section 2.4).
- We present the formal semantics of a core language of JEff, called Featherweight JEff (FJEff) (Section 3).
- We present the formal type system of FJEff and discuss its soundness properties (Section 4).

The syntax and semantics of FJEff have been modeled using Redex [13], an embedded domain-specific language for mechanizing programming languages. The source code of the models is available online [1].

The paper is concluded with a discussion of open problems and directions for further research. We hope that JEff can contribute to a better understanding of effectful programming in the context of object-oriented languages without built-in notions of state, identity, or inheritance [7].

2 JEff: Programming with Objects and Effects

2.1 Introduction

JEff is a Java-like language where custom effects can be defined as effect interfaces. The implementation of these effects is provided by handler classes. In this section we will explore the main characteristics of JEff using some illustrative scenarios.

Like Java, JEff features both classes and interfaces. It has multiple inheritance of interfaces, and it features both subtyping and parametric polymorphism (generics). To focus on the core aspects of combining object-orientation with effect handling, JEff does not feature implementation inheritance for classes [2]. Furthermore, JEff is a side-effect free language: there is no mutation, I/O, exceptions, etc. These effects are to be provided by libraries of effect handlers that simulate such effects. Programmers can design and implement their own custom effects, and provide new handlers for existing ones.

Effect interfaces can be used to define the signature of the effect methods, such as `print` in the introduction. Effect methods are implemented in handler classes which provide the effect semantics. Effect methods can resume execution, transferring control back to the point where the effect method was called, using the special context variable **`there`** which denotes a special object conforming to the predefined interface `Resume` that defines a single method `resume`.

If an effect method does not resume, the suspended execution stack is ignored and execution proceeds at the point where the handler was installed using the **`with`**-construct.

The **`with`**-construct thus acts as a delimiter of the dynamic context in which effect invocations are handled. In order to specify what to do with the value that is produced after executing the body of the **`with`**-expression – in the manner of a wrapper – handler classes must implement the predefined `Handler` interface, whose single `return` method acts as the required wrapper.

JEff features a type and effect system that assigns types to expressions, methods, interfaces and classes. A method whose body calls an unhandled operation needs to be annotated with a type that declares the called effect method. The client of that method must, therefore, provide at some point a handler for that particular effect, similar to how checked exception declarations propagate in Java.

We now illustrate effectful progamming in JEff using the standard examples of exception handling and state.

2.2 Exception Handling

In JEff, an effect type is defined by declaring an effect interface. For instance, the following effect interface (indicated by the keyword **`eff`** on the effect method `raise`) defines the effect of raising an exception:

```
interface Raise { eff Nothing raise(String s) }
```

The `Raise` effect interface declares the effect method `raise`, as indicated by the **`eff`** keyword. This operation receives a string as an argument and returns an object of type `Nothing` (which represents the bottom type). In this case, the return type signifies that the `raise` method will never return.

The following code illustrates how the `raise` effect is triggered:

```
Int divide(Int x, Int y)@Raise =
  if (y != 0) x/y else Raise::raise("Division by zero")
```

The method signature of `divide` reflects its required effects in its signature through the `Raise` annotation. If the divisor is equal to zero, the method `divide` throws the exception. The syntax to call operations is the name of the type corresponding to the effect interface followed by two colons, the name of the operation, and the arguments.

[1] https://github.com/cwi-swat/jeff-model

[2] As a reference, in [2] authors report that the interaction between exceptions, a particular case of effects, and inheritance is non-trivial.

The code above shows how to trigger an effect, but not how to handle it. In order to provide an interpretation to the `raise` effect, a handler object must be installed using the **`with`**-construct. This handler object must be an instance of a class that implements the `Raise` interface. For instance, the following expression installs a handler `h` to handle the effects triggered by `divide`:

```
with (h) { divide(4, 0) }
```

The **`with`**-expression acts as a dynamic scoping construct so that the `raise` invocation becomes a method call on the handler object `h`.

Since all the effects invoked in the code within the context of a **`with`**-construct are eventually handled, the body of the **`with`**-construct evaluates to a value. Handlers may capture this value and transform it before it is returned as the result of the **`with`**-expression itself. This is realized by the requirement that all handler objects must be instances of classes that implement the predefined `Handler` interface:

Definition 2.1 (Handler)**.**

```
interface Handler<Out, In> { Out return(In in) }
```

The `Handler` interface declares a single `return` method that captures the result of the **`with`**-construct when no more effects are triggered in its body. The two type parameters `In` and `Out` capture the type of the body of the **`with`**-expression, and the type of the **`with`**-expression itself, respectively.

There is nothing special about the `Handler` interface; it simply functions as the interface between the **`with`**-construct and its body and context, similar to how the `Iterable` interface interacts with the **`for`**-construct in Java. Note however, that the `return` method must be pure, since it has no effect annotations.

Now let's look at a potential implementation of a handler for `Raise`. The following `DefaultRaise` class defines a handler for the `raise` effect which simply returns a default value in case of an exception. The default value is provided when the class is instantiated through the `x` field. This default value is used as the value of the **`with`**-expression in case its body raises an exception.

```
class DefaultRaise<T>(T x)
    implements Raise, Handler<T, T> {
  T return(T t) = t
  eff Nothing raise(String s) = this.x
}
```

Both type parameters of `Handler` coincide and correspond to the type parameter `T`. This means that if an object of class `DefaultRaise` is used as a handler in a **`with`**-expression, both the handled body of the expression and the **`with`**-expression should have the same type. The `return` method, in this case, is the identity function. Hence, if the body does not raise an exception, as in **`with`** `(new DefaultRaise<Int>(-1)){ divide (4, 2)}`, the `return` method of the handler object will be called with the value returned by the body. In this case the value of the **`with`**-expression will therefore be `2`.

Looking closely at the implementation of `raise`, however, makes it clear that effect methods are special, since the type of their body does not match the declared return type at all. In fact, in this case, the return type is `Nothing`, whereas the type of the body expression is `T`! The reason for this is that the declared return type corresponds to the type of value that will be sent back to the calling context upon resumption using `resume`. The type of the body of an effect method should always correspond to the `Out` type parameter of the `Handler` interface. Since in this case, the `raise` method does not resume, it simply returns a value of that type, the default value.

The `divide` method only refers to the `Raise` effect interface, so it can be run in the context of any number of handler implementations of the `Raise` interface. For instance, here is another implementation of the `Raise` interface, where the result of a computation is wrapped in a `Maybe` (option) type:

```
class MaybeRaise<T>()
    implements Raise, Handler<Maybe<T>, T> {
  Maybe<T> return(T t) = new Some<T>(t)
  eff Nothing raise(String s) = new None()
}
```

In `MaybeRaise`, the `In` type parameter corresponds to `T` but the `Out` type parameter, that is, the one that corresponds to the type of the **`with`**-expression, is `Maybe<T>`. In this case, the `return` method wraps the value resulting from the evaluation of the **`with`**-body in a `Some` object. Dually, the `raise` method produces the empty value **`new`** `None()`. Note again that the `None` object will be the result of the **`with`**-expression.

We have seen that the `Nothing` return type of `raise` means that the method will never "return", in other words, that it will never transfer control back to the point of invocation. In the next section we illustrate the scenario in which effect methods resume execution at the point where they were called.

2.3 Resuming After Handling: State

When the `raise` effect is triggered in the body of a **`with`**-expression, the normal flow of computation is aborted and evaluation proceeds at the level of the **`with`**-expression. Most effects, however, require resuming execution at the point where the effect was invoked, after handling. JEff realizes resumption through the special variable **`there`**. The **`there`** object is implicitly brought in scope when an effect method executes, just like **`this`** is available in all method executions.

The type of **there** is defined by the Resume interface:

Definition 2.2 (Resume)**.**

```
interface Resume<T, Out, In> {
  Out resume(T x, Handler<Out, In> h)
}
```

The resume-method accepts two arguments. The first argument represents the value that is sent back to the calling context as the result of the effect invocation. The second argument represents the (possibly new) handler to install for the remainder of the execution. The concrete types for the type parameters are inferred from the context, but note that the second argument always needs to be a Handler.

Figure 2 shows how the state effect can be defined in JEff. The State effect interface defines two operations: get, to retrieve the state, and put to update the state. The interface is generic in what is stored, and it abstracts from how state itself is represented.

The figure also shows the handler class TupleState, which implements the State interface and the Handler interface. Both get and put resume the computation using the **there** object. The method get resumes execution with the current state **this**.s, within the context of the current handler object **this**. Alternatively, put resumes with the unit value (), and installs a new handler by constructing a new TupleState object with the updated state x. Note how the type of the first argument to the resume method always corresponds to the declared return type of the effect methods. Finally, TupleState defines the return method from the Handler interface, wrapping the current state **this**.s and result of the **with**-body x in a tuple.

The State effect and the TupleState handler can be used as follows:

```
Unit countDown()@State<Int> =
  Int i = State<Int>::get();
  if (i >= 0) {
    State<Int>::put(i - 1);
    this.countDown();
  }

with (new TupleState<Int,Unit>(2)) {
  this.countDown()
}
// evaluates to Tuple(0, ())
```

The method countDown requires the State effect over integers, as witnessed by the annotation. It simply decreases the stored value until it is zero. This method can then be invoked by bringing TupleState handler in scope using **with**.

Note again that countDown is independent from any implementation of state, and only depends on the effect interface State, in the same way that the divide method was only dependent on the effect interface Raise, and not on any particular implementation. Decoupling the interface of effect operations from handler operations allows client code to be independent of concrete handlers.

```
interface State<T> {
  eff T get()
  eff Unit put(T x)
}

class TupleState<T,U>(T s)
    implements State<T>, Handler<Tuple<T,U>,U> {
  eff T get() = there.resume(this.s, this)
  eff Unit put(T x) = there.resume((), new TupleState(x))
  Tuple<T,U> return(U x) = new Tuple<T,U>(this.s, x)
}
```

Figure 2. The State effect interface and a handler implementation TupleState using tuples

For instance, consider the following handler for state, which maintains a history of updates:

```
class LogState<T, U>(List<T> log)
    implements State<T>, Handler<Tuple<List<T>, U>,U>> {
  eff T get() = there.resume(this.log.last(), this)
  eff Unit put(T x) =
    there.resume((), new LogState<T,U>(log.append(x)));
  Tuple<List<T>, U> return(U x) =
    new Tuple<List<T>, U>(this.log, x)
}
```

Note that the State interface is still implemented over type parameter T, but the handler itself now uses List<T> as its representation to save the history of values that have been assigned to the state. Calling countDown in the context of a LogState handler, as in **with** (**new** LogState<Int, Unit>(2)){ countDown()}, will evaluate to the value Tuple([2, 1, 0],()).

2.4 Structuring Effectful Interpreters

One of the benefits of object-oriented programming is open extensibility of data types [6]. Given an interface defining a data type, (third-party) programmers can add new representation variants to the type without changing (or even recompiling) existing code. One use case where this is valuable is extensible AST-based interpreters.

Figure 3 shows the definition of an Exp data type with five classes realizing different kinds of expressions. The Exp interface defines a single eval method, annotated with Store and Env (environment) effect types[3]. Each concrete class implements the Exp interface[4].

The eval method in Lit does not use any effect, since it simply returns the field v. The Var class, however, requires Env reader effect to lookup bindings for variables. The classes

[3]Example implementations of Env and Store can be found in Appendix D.
[4]The effect annotations in the implementation classes specify only the effects that are actually used in their body. This is valid due to JEff's definition of overriding.

```
interface Exp {
  Val eval()@Store,Env
}

class Lit(Val v) implements Exp {
  Val eval() = this.v
}

class Var(String x) implements Exp {
  Val eval()@Env = Env::get().lookup(this.x)
}
```

```
class Deref(Exp cell) implements Exp {
  Val eval()@Store = Store::get(cell.eval())
}

class Assign(Exp loc, Exp val) implements Exp {
  Val eval()@Store =
    Store::put(this.loc.eval(), this.val.eval())
}

class Let(String x, Exp v, Exp b) implements Exp {
  Val eval()@Env =
    with (Env::get().extend(x, v.eval())){
      b.eval()
    }
}
```

Figure 3. Modular interpreters

`Deref` and `Assign` use the `Store` effect (similar to `State` of Figure 2) to realize cell dereferencing and assignment, respectively. Finally, the `Let` class models a lexically scoped binding construct, by first obtaining the current environment, extending it with a binding for `x`, and providing it as context for the evaluation of the body `b`.

Even though the `Exp` interface fixes the effect privileges of all interpreters, the effect handling mechanism of JEff makes it unnecessary to accept and propagate stores and environments explicitly, which would be needed in, e.g., Java, even when they are not used. Note also that the set of effect privileges on `eval` must be recursively closed, since `eval` methods might call effectful methods on dependencies. In this example all dependencies receiving method calls (i.e., `cell`, `loc`, `val`, `v`, and `b`) are `Exp` objects themselves, so this is trivially satisfied.

The AST classes of Figure 3 could then be used with the following `run` method:

```
Tuple<Store<Val>,Val> run(Exp<Val> exp) =
  with (new Store<Val>()) {
    with (new Env<Val>()) { exp.eval() }
}

run(
  new Let<Val>("x", new Cell(0),
    new Assign<Val>(
      new Var<Val>("x"),
      new Lit<Val>(new Num(42)))))
// evaluates to
// Tuple(Store(0, Map(Cell(0) -> Num(42))), Num(42))
```

The `run` method receives an expression and evaluates `exp` in the context of a fresh store and environment. The result will be a tuple of the store and the result. Note that `run` is pure, since all effects of `eval` are handled. Note further that `Store` and `Env` act both as handlers for their respective effects as well as data containers for cell-value and name-value pairs respectively; it is perfectly fine for JEff effect handlers to have ordinary methods as well.

Although the effect signature of `eval` is not extensible itself, it is still possible to extend the code of Figure 3 with new AST classes, without changing any of the existing classes, and without having to modify `run`. Achieving the same at the level of effects remains an open research question (see, e.g., [9], for a discussion and non-effectful solution).

2.5 Ad Hoc Overloading of Effects

A key feature offered by JEff's effect system is that dispatch of an effect invocation happens through both subtype polymorphism and parametric polymorphism (generics). This means that multiple handlers for the same effect interface can be in scope for a fragment of code, and, more importantly, they can be distinguished as well, since the effect invoking code explicitly qualifies the invoked effect.

This unique feature of JEff is illustrated in Figure 4. The `ToStr` effect allows some value of type `X` to be converted to a string. Without going into details of implementing handlers for this effect, the figure shows three specializations of this effect: two sub-interfaces (`IntToStr` and `IntToHex`) – representing different ways of converting an integer to a string –, and a specialization for converting booleans to string (`BoolToStr`).

Figure 4 also shows two methods invoking `ToStr` effects. The method `main1`, requires abstract `ToStr` effects, instantiated for both `Int` and `Bool`; it returns the concatenation of converting both method parameters to string. So the same effect interface is required to be in scope, instantiated over different argument types.

Assuming we have handler implementations `AnIntToStr`, and `ABoolToStr`, the `main1` method can be invoked as follows:

```
interface ToStr<X> {
  eff String toString(X x);
}

interface IntToStr extends ToStr<Int> { }

interface IntToHex extends ToStr<Int> { }

interface BoolToStr extends ToStr<Bool> { }

// same effect, different type parameter
String main1(Int n, Bool b)@ToStr<Int>, ToStr<Bool>
  = ToStr<Int>::toString(n) + " "
  + ToStr<Bool>::toString(b)

// same effect, same type parameter
String main2(Int n)@IntToStr, IntToHex
  = IntToStr::toString(n) + ": "
  + IntToHex::toString(n)
```

Figure 4. `ToStr` effects

```
with (new AnIntToStr<String>()) {
  with (new ABoolToStr<String>()) {
    this.main1(42, true)
  }
} // ⇒ "42 true"
```

The invocation `toString(n)` will dispatch to `AnIntToStr`, because `AnIntToStr` is a subtype of `ToString<Int>`. Similarly, `toString(b)` will dispatch to `ABoolToStr`, because `ABoolToStr` is a subtype of `ToString<Bool>`. Note, however, that `main1` only refers to `ToString`, so still benefits from subtype polymorphism. For instance, a handler implementation of `IntToHex` could also be installed to adapt the behavior of `main1` from the outside.

The three interfaces of Figure 4 allow us to go even one step further, and distinguish between different handlers over the same effect interface and argument type(s). This is illustrated in method `main2`.

In this case, the method prints out a single number using different presentations, the default one (`IntToStr`), and another one, in this case `IntToHex`. Again, assuming two handler implementations of these respective interfaces, allows `main2` to be invoked as follows:

```
with (new AnIntToStr<String>()) {
  with (new AnIntToHex<String>()) {
    this.main2(42)
  }
} // ⇒ "42 : 2A"
```

Again, the combination of subtyping and generic type instantiation provide additional flexibility in effectful programming.

3 Dynamic Semantics

In this section we present the semantics of Featherweight JEff (FJEff), a core calculus focusing on JEff's more distinctive semantic characteristics.

3.1 Syntax

$$
\begin{array}{lcl}
T, S, U, V, W & ::= & X \mid N \\
N, P, Q & ::= & C\langle \overline{T} \rangle \\
L & ::= & \texttt{class}\ C\langle \overline{X} \triangleleft \overline{N} \rangle (\overline{T}\,\overline{f}) \triangleleft \overline{N} \{ \overline{M} \} \\
 & \mid & \texttt{interface}\ C\langle \overline{X} \triangleleft \overline{N} \rangle \triangleleft \overline{N} \{ \overline{H} \} \\
\Xi & ::= & \overline{N} \\
H & ::= & [\texttt{eff}]\langle \overline{X} \triangleleft \overline{N} \rangle\ T\ m(\overline{T}\,\overline{x})@\,\Xi \\
M & ::= & H{=}e \\
e, d & ::= & x \mid e.f \mid e.\langle \overline{T} \rangle m(\overline{e}) \mid \texttt{new}\, N(\overline{e}) \\
 & \mid & N ::\langle \overline{T} \rangle m(\overline{e}) \mid \texttt{with}\,(e)\,\{\,e\,\} \\
v, w & ::= & \texttt{new}\, N(\overline{v}) \mid \texttt{new Resume}\,\{\texttt{resume}(x, x) = e\}
\end{array}
$$

Figure 5. FJEff syntax

The grammar of FJEff is shown in Figure 5. Metavariables B, C and D range over class and interface names; f and g over field names; m over method names; X and Y over type variables; and finally x and y over variables, including the special variables **`this`** and **`there`**. Comma-separated sequences are represented by overlined symbols, for example $\overline{M}$ represents a sequence of method declarations. Consecutive sequences represent sequencing tuples of elements as in $\overline{C}\ \overline{f}$ for field declarations.

Types can be either type variables X or non-variable types N. A type definition L can be either a class or an interface definition. Class definitions consist of the class's name, an optional list of bounded type parameters $X \triangleleft N$, followed by a possibly empty list of fields, an optional list of implemented interfaces (preceded by $\triangleleft$), and finally a list of methods.

An interface declaration follows the same structure, but does not feature fields and may only contain method headers. An effect set Ξ is a list of non-variable types where order is irrelevant. Method headers H may be marked as being an effect method using the keyword **`eff`** and consist further of the return type T, the method name m, the list of formal parameters $\overline{C}\ \overline{x}$, and an optional sequence of effect annotations Ξ. Method definitions M consists of a header and a body expression `e`.

Expressions e can be a variable, field reference, method invocation, object instantiation, effect method invocation, or a **`with`**-expression. The first four are standard. An effect method call consists of the type of an interface or class that defines the effect method, followed by two colons and then the name of the method and the arguments. The handling expression **`with`** contains two sub-expressions. The first one corresponds to the (handler) object that will handle (some of) the effects that will be be triggered during execution of the

$$
\begin{array}{rcl}
E & ::= & [] \mid E.f \mid E.\!<\overline{T}\!>\!m(e\ldots) \mid v.\!<\overline{T}\!>\!m(v\ldots E\,e\ldots) \mid \texttt{new}\,N(v\ldots E\,e\ldots) \mid N ::\!<\overline{T}\!>\!m(v\ldots E\,e\ldots) \\
 & \mid & \texttt{with}\,(E)\,\{\,e\,\} \mid \texttt{with}\,(v)\,\{\,E\,\} \\
X_N & ::= & [] \mid X_N.f \mid X_N.\!<\overline{T}\!>\!m(e\ldots) \mid v.\!<\overline{T}\!>\!m(v\ldots X_N\,e\ldots) \mid \texttt{new}\,N(v\ldots X_N\,e\ldots) \mid N ::\!<\overline{T}\!>\!m(v\ldots X_N\,e\ldots) \\
 & \mid & \texttt{with}\,(X_N)\,\{\,e\,\} \mid \texttt{with}\,(\texttt{new}\,Q(v\ldots))\,\{\,X_N\,\} \qquad \text{when } \bullet \vdash Q \not<: N
\end{array}
$$

Figure 6. Evaluation contexts for reduction semantics

$$\frac{\mathit{fields}(N)=\overline{T}\,\overline{f}}{\texttt{new}\,N(\overline{v}).f_i \longrightarrow v_i}\ \textsc{r\_field}$$

$$\frac{\mathit{mbody}(m\!<\!\overline{V}\!>, N) = \overline{x}.e}{\texttt{new}\,N(\overline{v}).\!<\!\overline{V}\!>\!m(\overline{w}) \longrightarrow [\texttt{new}\,N(\overline{v})/\texttt{this}, \overline{w}/\overline{x}]e}\ \textsc{r\_invk}$$

$$\frac{}{\texttt{new}\,\texttt{Resume}\,\{\texttt{resume}(x_v,\ x_h)\ =\ e\}.\texttt{resume}(v_1,\ v_2) \longrightarrow [v_1/x_v, v_2/x_h]e}\ \textsc{r\_resume}$$

$$\frac{}{\texttt{with}\,(\texttt{new}\,N(\overline{v}))\{v\} \longrightarrow \texttt{new}\,N(\overline{v}).\texttt{return}(v)}\ \textsc{r\_return}$$

$$\frac{\bullet \vdash Q\!<\!:N \qquad \mathit{mbody}(m\!<\!\overline{V}\!>, Q)\!=\!\overline{x}.e \qquad v_k = \texttt{new}\,\texttt{Resume}\,\{\texttt{resume}(x_v,\ x_h) = \texttt{with}\,(x_h)\,\{\,X_N[x_v]\,\}\,\}}{\texttt{with}\,(\texttt{new}\,Q(\overline{v}))\,\{\,X_N[N::m\!<\!\overline{V}\!>\!(\overline{w})]\,\} \longrightarrow [\texttt{new}\,Q(\overline{v})/\texttt{this}, v_k/\texttt{there}, \overline{w}/\overline{x}]e}\ \textsc{r\_eff\_invk}$$

Figure 7. Reduction rules

body. Finally, objects v, w correspond to a fully-evaluated instantiation expressions with **new**, or resumption object containing a definition of the `resume` method as defined by the `Resume` interface (see Definition 2.2).

There is an assumed fixed class table CT representing a mapping from class and interface names to their declarations. The class table needs to satisfy some sanity conditions in the spirit of [8]. We do not elaborate on these conditions since they are standard and not central to our discussion.

3.2 Reduction Semantics

We present the operational semantics of FJEff using Felleisen-style evaluation contexts [23]. We use two evaluation contexts, E and X_N, shown in Figure 6.

The E context is the usual context for call-by-value evaluation, while the X_N context is used for the context that a handler delimits. It follows the same structure as E except that it matches **with**-expressions so that the body X_N does not contain **with**-expressions which handle effect N. In other words, it captures the nearest enclosing **with**-expression that is able to handle effect N. The side-condition ensures this by disallowing Q to be a subtype of N.

Figure 7 shows the evaluation rules of FJEff. The semantics uses two auxiliary lookup functions *fields* (to map field names to field indices) and *mbody* (to obtain the body of a method); their definition is included in Appendix A.

Rule R_FIELD defines field lookup. It maps a field name to its position in the sequence of values in an object using the *fields* lookup function. Rule R_INVK defines the semantics for ordinary method invocation by obtaining the body of the method m using the *mbody* lookup function. The expression then reduces to the method body e with substitutions applied for **this** and the formal parameters $\overline{x}$. The rule R_RESUME is similar to R_INVK but works on synthesized resumption objects.

There are two cases for **with**-expressions. The first one, R_RETURN, deals with the case in which the body expression has been fully evaluated to a value. In that case the expression reduces to a `return` invocation expression on the handler object.

The second rule R_EFF_INVK applies when **with**-bodies contain remaining effect method invocations, and thus implements effect dispatch. The context X_N ensures that the Q object is the directly enclosing handler servicing the effect N. The effect method m is looked up in the Q object, and the **with**-expression is reduced to its body e with substitutions applied for **this**, **there**, and the formal parameters $\overline{x}$.

The special variable **there** is substituted for a resumption object v_k whose `resume` method installs the handler x_h and continues execution of context X_N with x_v plugged in as the result of the effect method. As a consequence, the value of an effect method invocation at the call site will be x_v whenever the effect method resumes.

Compared to other calculi for effects and handlers that are of functional nature, e.g. [14, 15], FJEff's rule for effect dispatch is special in that the syntax for the invocation includes an effect qualifier N that enables effect selection using subtyping and parametric polymorphism, as it has been discussed in the `ToStr` example of Section 2.5. In R_EFF_INVK, it is clear that the handler Q that is selected among those in the runtime stack, is the one that is a subtype of the type specified by qualifier N. The `ToStr` example illustrates the

$$\frac{\Delta = \overline{X}{<:}\overline{N},\ \overline{Y}{<:}\overline{P} \qquad \Xi_m\ ;\ \Delta\ ;\ \overline{x}{:}\overline{T},\ \texttt{this}{:}C{<}\overline{X}{>} \vdash e_0 : S \qquad \Delta \vdash S{<:}T}{<\overline{Y}\triangleleft\overline{P}> T\ m(\overline{T}\ \overline{x})\ @\ \Xi_m\ \ \text{OK IN}\ \ C{<}\overline{X}\triangleleft\overline{N}{>}} \quad \text{T\_METH\_REG}$$
$$<\overline{Y}\triangleleft\overline{P}> T\ m(\overline{T}\ \overline{x})\ @\ \Xi_m = e_0\ \ \text{OK IN}\ \ C{<}\overline{X}\triangleleft\overline{N}{>}$$

$$\frac{\begin{array}{c}\bullet \vdash C{<}\overline{X}{>} {<:} \texttt{Handler}{<}T_{\text{out}}, T_{\text{in}}{>} \qquad \Delta = \overline{X}{<:}\overline{N},\ \overline{Y}{<:}\overline{P} \\ \Xi_m\ ;\ \Delta\ ;\ \overline{x}{:}\overline{T},\ \texttt{this}{:}C{<}\overline{X}{>},\ \texttt{there}{:}\texttt{Resume}{<}T, T_{\text{out}}, T_{\text{in}}{>} \vdash e_0 : S \qquad \Delta \vdash S{<:}T_{\text{out}} \\ \texttt{eff}\ {<}\overline{Y}\triangleleft\overline{P}{>}\ T\ m(\overline{T}\ \overline{x})\ @\ \Xi_m\ \ \text{OK IN}\ \ C{<}\overline{X}\triangleleft\overline{N}{>}\end{array}}{\texttt{eff}\ {<}\overline{Y}\triangleleft\overline{P}{>}\ T\ m(\overline{T}\ \overline{x})\ @\ \Xi_m = e_0\ \ \text{OK IN}\ \ C{<}\overline{X}\triangleleft\overline{N}{>}} \quad \text{T\_METH\_EFF}$$

Figure 8. Method typing rules

$$\frac{}{\Xi;\Delta;\Gamma \vdash x : \Gamma(x)} \quad \text{T\_VAR}$$

$$\frac{\Xi;\Delta;\Gamma \vdash e_0 : T_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(T_0)) = \overline{T}\ \overline{f}}{\Xi;\Delta;\Gamma \vdash e_0.f_i : T_i} \quad \text{T\_FIELD}$$

$$\frac{\Delta \vdash N\ \text{ok} \qquad \mathit{fields}(N) = \overline{T}\ \overline{f} \qquad \Xi;\Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S}{<:}\overline{T}}{\Xi;\Delta;\Gamma \vdash \texttt{new}\ N(\overline{e}) : N} \quad \text{T\_NEW}$$

$$\frac{\begin{array}{c}\Xi;\Delta;\Gamma \vdash e_0 : T_0 \qquad \mathit{mtype}(m,\ \mathit{bound}_\Delta(T_0)) = {<}\overline{Y} \triangleleft \overline{P}{>}\overline{U}{\to}U@\Xi_m \qquad \Delta \vdash \Xi \preceq \Xi_m \\ \Delta \vdash \overline{V}\text{ok} \qquad \Delta \vdash \overline{V}{<:}[\overline{V}/\overline{Y}]\overline{P} \qquad \Xi;\Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S}{<:}[\overline{V}/\overline{Y}]\overline{U}\end{array}}{\Xi;\Delta;\Gamma \vdash e_0.m{<}\overline{V}{>}(\overline{e}) : [\overline{V}/\overline{Y}]U} \quad \text{T\_INVK}$$

$$\frac{\begin{array}{c}\mathit{mtype}(m,\ N) = \texttt{eff}\ {<}\overline{Y} \triangleleft \overline{P}{>}\ \overline{U}{\to}U@\Xi_m \qquad \Delta \vdash \Xi \preceq \Xi_m \qquad \Delta \vdash \Xi\ \preceq\ N \\ \Delta \vdash \overline{V}\text{ok} \qquad \Delta \vdash \overline{V}{<:}[\overline{V}/\overline{Y}]\overline{P} \qquad \Xi;\Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S}{<:}[\overline{V}/\overline{Y}]\overline{U}\end{array}}{\Xi;\Delta;\Gamma \vdash N{::}m{<}\overline{V}{>}(\overline{e}) : [\overline{V}/\overline{Y}]U} \quad \text{T\_EFF\_INVK}$$

$$\frac{\begin{array}{c}\Xi;\Delta;\Gamma \vdash e_0 : T_0 \qquad \mathit{mtype}(\texttt{return},\ \mathit{bound}_\Delta(T_0)) = U_{\text{in}}{\to}U_{\text{out}} \qquad \mathit{bound}_\Delta(T_0), \Xi;\Delta;\Gamma \vdash e_1 : T_1 \\ \Delta \vdash \mathit{bound}_\Delta(T_0) {<:} \texttt{Handler}{<}U_{\text{out}}, U_{\text{in}}{>} \qquad \Delta \vdash T_1{<:}U_{\text{in}}\end{array}}{\Xi;\Delta;\Gamma \vdash \texttt{with}\ (e_0)\ \{\ e_1\ \} : U_{\text{out}}} \quad \text{T\_WITH}$$

Figure 9. Expression typing

consequences of this language design and the opportunities that become available in terms of new patterns for structuring effectful code, unavailable in functional languages with effects.

4 Type System

4.1 Introduction

JEff is a statically typed language with a nominal type system, where both classes and interfaces introduce types, arranged in a subtype lattice. In this section we present the type system of the simplified core language FJEff. The type system of FJEff is mostly standard, except that it ensures that JEff methods are effect-safe: whenever an effect is triggered it is either handled using a syntactically enclosing **`with`**-construct, or the enclosing method is annotated with a type defining the effect.

The type system further makes use of the `Handler` and `Resume` interfaces for checking effect method declarations, and the **`with`**-construct itself. Below we describe the type rules for method definitions and expressions[5]. The rules are similar to the typing rules used in Featherweight Generic Java [8] and employ auxiliary definitions for subtyping, well-formedness and overriding, included in Appendix B.

4.2 Method Typing

Figure 8 shows the two typing rules for method definitions in classes. The rule T_METH_REG checks the validity of ordinary, non-effect method declarations. The body of the method e_0 is type checked in context of the effect set Ξ_m, the type variable environment Δ (derived from the type parameters of the method and class), and an initial type environment defining the type of **`this`**. The set Ξ_m can be seen as the set of effect privileges available in the body of m. A method declaration is then valid if its header is valid and the type of e_0 is a subtype of the declared return type T.

The second rule defines the type correctness of effect methods in a similar fashion. The first difference, however, is that,

[5]Definitions for header, class and interface typing can be found in Appendix C.

in this case, C needs to implement the `Handler` interface. Second, the initial type environment also defines the type of the **`there`** variable in terms of the `Resume` interface. Finally, since the return type of an effect method corresponds to the type of the value used in resumptions, in this case the derived type of e_0 (S) must be a subtype of T_{out}, the return type of the `return` method defined by C.

4.3 Expression Typing

The rules for expression typing are shown in Figure 9. Here we highlight the most salient differences with respect to those used in Featherweight Generic Java. First of all, there are two different rules for checking method invocation: T_INVK for regular method invocation and T_EFF_INVK for effect method invocation. These rules enforce that regular methods can only be called using normal method invocation syntax, and effect methods only via effect call syntax by requiring that the method type returned by the *mbody* metafunction has the right effect annotation. Next to the type variable environment and the type environment, expressions are typed in the context of an effect privilege set Ξ, as introduced by the method typing rules shown in Figure 8.

For instance, the rule for ordinary method invocation T_INVK checks that the declared effects of m (Ξ_m) are included in the privilege set Ξ using the condition $\Delta \vdash \Xi \preceq \Xi_m$. The rule for effect invocation T_EFF_INVK performs the same check, but additionally enforces that the requested effect C is also in the privilege set Ξ. Dually, the rule for **`with`**-expressions (T_WITH) *extends* the privilege set for e_1 with the type of e_0 (T_0).

The relation $\preceq$ defines a preorder between two sets of types, and intuitively extends subtyping over sets of types. It is defined by first defining $\preceq$:

$$\Delta \vdash \Xi \preceq T \;\equiv\; \exists T' \in \Xi : \Delta \vdash T' <: T \qquad (1)$$

(Ξ type in Ξ *handles* type T)

The full relation is then obtained as follows:

$$\Delta \vdash \Xi_1 \preceq \Xi_2 \;\equiv\; \forall T \in \Xi_2 : \Delta \vdash \Xi_1 \preceq T \qquad (2)$$

(Ξ_1 *handles* all types in Ξ_2)

This relation is used in determining whether a privilege set is powerful enough to handle all effects requested by a certain expression. The relation is further used in checking the validity of method implementations, where the effect annotations of method definitions in a class may be less demanding according to $\preceq$ than the declared annotations in a declaring interface. In plain language this means that the effect payload of a method, i.e. the effects it might invoke, may be less than what is declared. Note in particular that pure method implementations (i.e. without any effects) conform to well-typed interface method declarations with arbitrary sets of effect annotations, because $\Xi \preceq \bullet$.

4.4 Soundness

Soundness is often stated as "well-typed programs cannot go wrong". We sketch the proof of soundness using progress and preservation.

For progress, we make a distinction by considering that expressions in normal form are not only values but also expressions $X_N[N::m<V>(\overline{e})]$ whose next evaluation step requires the handling of an operation call. By pairing the latter with a constraint to a set of effect privileges Ξ, we have a new class of expressions that we do not consider stuck.

Definition 4.1 (Normal Form)**.** An expression e is in normal form with respect to Ξ if either (a) is a value v, or (b) is an expression of the form $X_N[N::m<V>(\overline{e})]$ such that $\bullet \vdash \Xi \preceq N$.

Definition 4.2 (Non-stuckness)**.** An expression e is non-stuck with respect to Ξ if either (a) is in normal form with respect to Ξ, or (b) there is an e', such that $e \to e'$.

Lemma 4.3 (Progress)**.** *If* $\Xi; \bullet; \bullet \vdash e : T$*, then* e *is non-stuck with respect to* Ξ*.*

Proof sketch. By induction on the structure of type derivations, with a case analysis on the last rule used. The interesting case is T_WITH, in particular when e corresponds to **`with`** `(new Q(`$\overline{v}$`)){`e_1`}`. By rule T_WITH, we have $Q, \Xi; \bullet; \bullet \vdash e_1 : T_1$ for some T_1. Then, by induction hypothesis, e_1 is non-stuck, thus, it is either (a) a redex, in which case e progresses; (b) a value, in which case rule R_RETURN applies; or (c) of the form $X_N[N::m<V>(\overline{e})]$, such that $\bullet \vdash Q, \Xi \preceq N$, in which case, by the definition of $\preceq$ (cf. Section 4.3), either: (c.1) $Q <: N$, which implies that rule R_EFF_INVK applies; or (c.2) $Q \not<: N$, which implies that $\Xi \preceq N$; and because of this together with the fact that no reduction rules apply, e fits in the definition of a context X_N, being of the form $X_N[N::m<V>(\overline{e})]$, and thus, is in normal form with respect to Ξ (therefore non-stuck with respect to Ξ).

Lemma 4.4 (Preservation)**.** *If* $\Xi; \Delta; \Gamma \vdash e : T$ *and* $e \to e'$*, then* $\Xi; \Delta; \Gamma \vdash e' : T'$ *for some* T' *such that* $\Delta \vdash T' <: T$*.*

Proof sketch. By induction over the reduction rules. A number of necessary lemmas are needed, such as that (1) term and (2) type substitution preserve typing, that (3) subtyping preserves method typing, and that (4) method bodies conform to declared return types. Their proofs are similar to those found in [8]. The replacement lemma (5) states that if there is a deduction D ending in $\Gamma \vdash C[e] : T$, where C is a context; and there is a sub-deduction D' ending in $\Gamma' \vdash e : T'$, and $\Gamma \vdash e' : T'$, then $\Gamma \vdash C[e'] : T$ (proof is similar to replacement in [23]).

The interesting cases are rules R_EFF_INVK, R_RETURN and R_RESUME. The crucial facts for R_EFF_INVK are:

(a) We know by T_WITH that $X_N[N::m<V>(\overline{v})]$ has type T_1 for some T_1. We also know trivially that $N::m<V>(\overline{v})$ has type S for some S. By letting variable x_v have

type S and plugging x_v inside context $X_N[]$ in place of the effect method call, we obtain expression $X_N[x_v]$, which by (5) retains type T_1. By letting x_h have type T_0 for some T_0<:$Handler$<U_{out}, U_{in}>, we know that expression **with**(x_h){$X_N[x_v]$} in the body of the built-in resumption object has type U_{out} and thus resumption object v_k conforms by construction to interface `Resume`<S, U_{out}, U_{in}>, as required by the specification of resumption objects.

(b) By premise $\bullet \vdash Q <: N$ and (3) together with (4), we know that the expression implementing an effect method call N::m<V>($\overline{v}$) is to be found in the corresponding method m in handler class Q.

Rule R_RESUME relies on the fact that resumption objects are not expressible and are only introduced via substitution of **there** in R_EFF_INVK. From (a), we know that the resumption object is well-typed to `Resume`<S, U_{out}, U_{in}> for some S, U_{in} and U_{out}, and then the argument continues as the one for standard method invocation.

In case of R_RETURN, since N<: `Handler`<U_{out}, U_{in}> for some U_{out} and U_{in}, and `Handler` defines `return` to have the type $U_{in} \rightarrow U_{out}$, both the **with** expression and the result of invoking `return` have the same type.

5 Discussion & Related Work

5.1 Objects for Effect

According to Cook the essence of objects is encapsulation and dynamic dispatch [7]. These are precisely the aspects that we have leveraged in JEff for supporting effectful programming. This can be seen from the fact that, apart from the **with**-construct, all other parts of effect handlers are realized by calling methods, all of them defined in interfaces or classes.

Effect operations are defined as methods with ordinary type signatures, but with bodies typed according to the `Handler` interface. Effect resumption is method invocation on the special **there** object, which is typed by the ordinary `Resume` interface. The **with**-construct brings `Handler` objects into dynamic scope, and when its syntactic body has evaluated to a value, the result is passed through the ordinary `return` method as required by `Handler`. Both the `Resume` and `Handler` interfaces are not special, but simply part of JEff's standard library. Like Java's `Closeable` and `Iterable`, they merely provide the interface between certain language features (in JEff's case **with** and **there**) and the objects defined by the programmer.

5.2 Effect Polymorphism

Effect polymorphism refers to the ability of code to operate on objects with varying effect surfaces, where the actual effectfulness of code derives from dependencies such as method parameters or fields. In JEff, all method declarations – both in interfaces and classes – need to be annotated with concrete effect types for unhandled effects. As a result, JEff does not support effect polymorphism.

Consider the two interfaces below, `Function` and `List`:

```
interface Function<T,U> {
  U apply(T t) // pure
}

interface List<T> {
   <U> List<U> map(Function<T,U> f) // pure
}
```

Both the `apply` method in `Function`, and `map` in `List` have no effect annotations. As a result, implementations of these methods are required to be pure. Another consequence is that the argument to `map` must be pure as well, and hence `map` is effect monomorphic: it only applies pure functions to the list. Since it is not possible to abstract over the effect signature of a (set of) method(s), different `map`s are needed for `Functions` with different effect payloads.

A similar effect can be observed in the modular interpreters presented in Section 2.4. Although new `Exp`s can be defined in a modular fashion, the allowed effects of the `eval` method are determined and fixed in the `Exp` interface. With a mechanism for effect polymorphism, this could potentially be made more flexible, where the `eval` in `Exp` would be polymorphic, and each implementation would carry its own effect signature.

Supporting effect polymorphic methods in a language like JEff is challenging. Existing languages like Frank [15] and Koka [14] support effect polymorphic functions but have significantly different type systems than JEff. For instance, Koka's row polymorphism allows the effect payload of a function to be left partially open; unification is then used to add rows to the types during type inference. It is however unclear how to port this kind of inference to object-oriented languages.

A possible middle-ground solution is presented by Toro and Tanter [20] in the form of a gradual polymorphic effect system for Scala. By giving up some static guarantees about the effectfulness of code, the use of higher-order functions like `map` can be made more flexible. In this case both `apply` and `map` would be annotated with the special `@unknown` annotation (denoted as ¿ in [3]), which signifies that there is no static information about the effects of `map`. Implementations of such methods can provide the missing information with concrete annotations. The type system can then derive more specific effect payloads at concrete call sites; if it cannot, then dynamic checks ensure that the effect will be handled correctly.

5.3 Propagation of Annotations

JEff's effect propagation mechanism is similar to Java's checked exceptions and, as such, suffers from the same problems. Programmers need to annotate each method with its allowed

effects which is verbose and makes code less flexible. An interface method declares a number of effects but the corresponding method in a class implementing this interface might require a new effect. In that case the annotations of the parent interface should be modified to incorporate the new effect, and consequently, all classes implementing it.

The lack of flexibility of checked annotations has been addressed in the work on anchored exceptions, where method call dependencies are taken into account when propagating exceptions in `throws`-clauses [21]. Based on this work and in the context of Scala, Ritz proposes Lightweight Polymorphic Effects (LPE) [19] as an attempt to generalize the annotation-based style of checked exceptions to a wider range of effects. LPE allows programmers to annotate effectful Scala code with effect annotations, where effects are defined using a customizable effect lattice, inspired by [16]. We consider incorporating a similar mechanism to JEff as future work.

5.4 Related Work

Algebraic effects [17, 18] are a mechanism to represent effects in functional languages. An effect defines the signature of a set of operations. The actual semantics of an effect is provided by handlers, dynamically scoped constructs that implement the behavior for each operation.

In JEff, effect interfaces are analogous to effect signatures, while handler classes correspond to their implementation. The handler abstraction is further discussed in [10], where a formal definition together with library-based implementations in several languages is presented. This work also introduces the distinction between deep and shallow handlers. Deep handlers automatically wrap the continuation within the current handler. Shallow handlers assume no handling by default, and therefore require the programmer to install a new handler manually if so desired. JEff is closer in spirit to the latter, since we require a handler object as the second argument to **`there`**`.resume`.

Besides the handler libraries presented in [10], there are several other library-based encoding of effects and handlers [5, 11, 12, 24]. Compared to library-based encodings, the built-in effects in JEff have the following advantages:

- Reducing the boilerplate caused by the accidental complexity of the effects embedding.
- Having a clearer computation model of the interaction between object-oriented concepts and effects. For instance, the Effekt library [5] encodes algebraic effects using sophisticated Scala features, such as implicit function types. In JEff, the interactions are clear and rely on a limited number of concepts captured by the FJEff calculus.
- Opening the door to domain-specific compiler optimizations taking into account the native representation of effects. For example, in [14], Leijen shows an efficient compilation of effect handling using a type-directed selective CPS translation.

Besides the library-based approaches, we have discussed a number of functional languages that provide native support for handlers and effects, using however different mechanisms for effect propagation.

Koka [14], for example, features an effect inference system that requires minimal annotations from users by relying on the polymorphic row types discussed in the previous section.

Frank [15], on the other hand, treats function application as a special case of a more generic mechanism of operators that act as interpreters of effects. Rather than accumulating effects outwards via type inferencing as in Koka, effects are propagated inwards using an ambient ability, similar to the privilege sets Ξ used in JEff.

6 Conclusion

Effect handlers are a technique to define, scope and modularize side-effects in programming languages that do not support them natively. While originally introduced and explored in the context of (purely) functional programming languages, it is an open question how effect handling could be supported first-class in an object-oriented programming language. In this paper we presented first steps towards answering this question, in the form of JEff, a purely object-oriented language with built-in support for effectful programming. Effects are defined using effect methods which obey special typing rules and have access to the current continuation for resuming computation. Handler objects are brought into dynamic scope using the **`with`**-construct; effects that are not handled need to be declared at the method level, similar to Java's `throws`-clause.

We have shown how common effects, like exception handling and state, can be defined within JEff, and how effects can be used to structure extensible interpreters. Furthermore the `ToStr` example illustrated how the type qualifier of effect invocations enable ad hoc overloading of effects. The semantics of JEff are formalized based on a core subset, FJEff, including a type system that ensures that all effects are properly handled or propagated. Finally, we provided intuitions that show that the type system is sound with respect to the semantics.

As directions for further work, we consider mechanizing the soundness proof, implementing the language, and extending JEff with support for (implementation) inheritance. In particular, support for `super`-calls would allow programmers to customize effect handlers. The biggest open question, however, is how to reconcile the open-world assumptions of object-orientation with effect handling. Modular extensibility of both data types and operations has been solved by solutions to the expression problem [22]; the next question is how to realize the same at the level of effects. JEff represents the first steps towards better understanding this question from the perspective of object-oriented programming.

A Auxiliary Definitions

Field lookup:

$$\frac{\texttt{class}\ C\langle\overline{X}\triangleleft\overline{N}\rangle(\overline{S}\,\overline{f})\triangleleft\overline{P}\{\ldots\}}{\mathit{fields}(C\langle\overline{T}\rangle) = [\overline{T}/\overline{X}]\overline{S}\,\overline{f}}\ \text{F\_CLASS}$$

Method type lookup:

$$\frac{\texttt{class}\ C\langle\overline{X}\triangleleft\overline{N}\rangle(\overline{S}\,\overline{f})\triangleleft\overline{P}\{\overline{M}\} \qquad [\texttt{eff}]\langle\overline{Y}\triangleleft\overline{Q}\rangle\ U\ m(\overline{U}\,\overline{x})@\Xi_m{=}e\ \in\ \overline{M}}{mtype(m, C\langle\overline{T}\rangle) = [\texttt{eff}]\,[\overline{T}/\overline{X}](\langle\overline{Y}\triangleleft\overline{Q}\rangle\overline{U}\to U@\Xi_m)}\ \text{MT\_CLASS}$$

Method body lookup:

$$\frac{\texttt{class}\ C\langle\overline{X}\triangleleft\overline{N}\rangle(\overline{S}\,\overline{f})\triangleleft\overline{P}\{\overline{M}\} \qquad [\texttt{eff}]\langle\overline{Y}\triangleleft\overline{Q}\rangle\ U\ m(\overline{U}\,\overline{x})@\Xi_m{=}e\ \in\ \overline{M}}{mbody(m\langle\overline{V}\rangle, C\langle\overline{T}\rangle) = \overline{x}.[\overline{T}/\overline{X}, \overline{V}/\overline{Y}]e}\ \text{MB\_CLASS}$$

B Subtyping and Type Well-formedness Rules

Bound of type:

$$bound_\Delta(X) = \Delta X \qquad bound_\Delta(N) = N$$

Subtyping:

$$\frac{}{\Delta \vdash T <: T}\ \text{S\_REFL} \qquad \frac{}{\Delta \vdash X <: \Delta(X)}\ \text{S\_VAR} \qquad \frac{\Delta \vdash S <: T \qquad \Delta \vdash T <: U}{\Delta \vdash S <: U}\ \text{S\_TRANS}$$

$$\frac{\texttt{class}\ C\langle\overline{X}\triangleleft\overline{N}\rangle(\ldots)\triangleleft\overline{P}\{\ldots\} \qquad Q \in \overline{P}}{\Delta \vdash C\langle\overline{T}\rangle <: [\overline{T}/\overline{X}]Q}\ \text{S\_CLASS\_M} \qquad \frac{\texttt{interface}\ C\langle\overline{X}\triangleleft\overline{N}\rangle\triangleleft\overline{P}\{\ldots\} \qquad Q \in \overline{P}}{\Delta \vdash C\langle\overline{T}\rangle <: [\overline{T}/\overline{X}]Q}\ \text{S\_IFACE}$$

Well-formed types:

$$\frac{}{\Delta \vdash \texttt{Object}\ \text{ok}}\ \text{WF\_OBJECT} \qquad \frac{X \in dom(\Delta)}{\Delta \vdash X\ \text{ok}}\ \text{WF\_VAR}$$

$$\frac{\texttt{class}\ C\langle\overline{X}\triangleleft\overline{N}\rangle(\ldots)\triangleleft\overline{P}\{\ldots\} \qquad \Delta \vdash \overline{T}\ \text{ok} \qquad \Delta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{N}}{\Delta \vdash C\langle\overline{T}\rangle\ \text{ok}}\ \text{WF\_CLASS}$$

$$\frac{\texttt{interface}\ C\langle\overline{X}\triangleleft\overline{N}\rangle\triangleleft\overline{P}\{\ldots\} \qquad \Delta \vdash \overline{T}\ \text{ok} \qquad \Delta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{N}}{\Delta \vdash C\langle\overline{T}\rangle\ \text{ok}}\ \text{WF\_IFACE}$$

Predefined interfaces:

```
interface Object { }

interface Handler<Out, In> { Out return(In in) }

interface Resume<T, Out, In> { Out resume(T x, Handler<Out, In> h) }
```

Valid method overriding:

$$\frac{mtype(m, N) = \langle\overline{Z}\triangleleft\overline{Q}\rangle\,\overline{U}\to U_0\,@\,\Xi_0 \ \text{ implies } \ \overline{P}, \overline{T} = [\overline{Y}/\overline{Z}](\overline{Q}, \overline{U}) \ \text{ and } \ T_0 = [\overline{Y}/\overline{Z}]U_0 \ \text{ and } \ \overline{Y}{<:}\overline{P} \vdash [\overline{Y}/\overline{Z}]\,\Xi_0 \preceq \Xi}{override(m, N, \langle\overline{Y}\triangleleft\overline{P}\rangle\,\overline{T}\to T_0\,@\,\Xi)}$$

C Header, Class and Interface Typing

Method header typing:

$$\frac{\texttt{class}\ C{<}\overline{X}{\triangleleft}\overline{N}{>}(\ldots){\triangleleft}\ Q_1 \ldots Q_n\{\ldots\} \qquad \overline{X}{<:}\overline{N},\ \overline{Y}{<:}\overline{P} \vdash \overline{T},\ T,\ \overline{P},\ \Xi_m\ \text{ok} \qquad override(m,\ Q_1,\ {<}\overline{Y}{\triangleleft}\overline{P}{>}\overline{T}{\to}T@\Xi_m) \ldots override(m,\ Q_n,\ {<}\overline{Y}{\triangleleft}\overline{P}{>}\overline{T}{\to}T@\Xi_m)}{[\texttt{eff}]\ {<}\overline{Y}{\triangleleft}\overline{P}{>}\ T\ m(\overline{T}\ \overline{x})@\Xi_m \quad \text{OK IN} \quad C{<}\overline{X}{\triangleleft}\overline{N}{>}} \ \text{T\_HEADER\_CLASS}$$

$$\frac{\texttt{interface}\ C{<}\overline{X}{\triangleleft}\overline{N}{>}{\triangleleft}\ Q_1 \ldots Q_n\{\ldots\} \qquad \overline{X}{<:}\overline{N},\ \overline{Y}{<:}\overline{P} \vdash \overline{T},\ T,\ \overline{P},\ \Xi_m\ \text{ok} \qquad override(m,\ Q_1,\ {<}\overline{Y}{\triangleleft}\overline{P}{>}\overline{T}{\to}T@\Xi_m) \ldots override(m,\ Q_n,\ {<}\overline{Y}{\triangleleft}\overline{P}{>}\overline{T}{\to}T@\Xi_m)}{[\texttt{eff}]\ {<}\overline{Y}{\triangleleft}\overline{P}{>}\ T\ m(\overline{T}\ \overline{x})@\Xi_m \quad \text{OK IN} \quad C{<}\overline{X}{\triangleleft}\overline{N}{>}} \ \text{T\_HEADER\_IFACE}$$

Class and interface typing:

$$\frac{\overline{X}{<:}\overline{N} \vdash \overline{N},\ \overline{T},\ \overline{P}\ \text{ok} \qquad \overline{M}\ \text{OK IN}\ C{<}\overline{X}{\triangleleft}\overline{N}{>}}{\texttt{class}\ C{<}\overline{X}{\triangleleft}\overline{N}{>}(\overline{T}\ \overline{f}){\triangleleft}\ \overline{P}\{\overline{M}\}} \ \text{T\_CLASS}$$

$$\frac{\overline{X}{<:}\overline{N} \vdash \overline{N},\ \overline{P},\ \text{ok} \qquad \overline{H}\ \text{OK IN}\ C{<}\overline{X}{\triangleleft}\overline{N}{>}}{\texttt{interface}\ C{<}\overline{X}{\triangleleft}\overline{N}{>}{\triangleleft}\ \overline{P}\{\overline{H}\}} \ \text{T\_IFACE}$$

D Env and Store

The following code provides implementations of the `Env` and `Store` types referred to in Section 2.4. Both `Env` and `Store` use an immutable `Map` class. Note how both `Env` and `Store` use the `DefaultRaise` handler of Section 2.2 to deal with missing keys. To make client code less unwieldy, JEff features default values for fields as notational short-hand; since object construction is always pure in JEff, the only allowed expressions as default values are object constructor calls with **`new`** or literals. Notice too that both the `Env` and the `Store` classes are a data structure and a handler at the same time. In particular, they represent a domain-specific type of handler, that in this case is suitable for the "interpretation of expressions" domain. Because of this, we have fixed the incoming type of the handler to `Val`, which brings as consequence that any **`with`**-expression installing these handlers will enclose a `Val`-producing expression.

```
class Map<T, U>() {
  U get(T t)@Raise = ...
  Map<T, U> put(T t, U u) = ...
}

class Env(Map<String,Val> map = new Map<String,Val>())
    implements Handler<Val, Val> {

  eff Env get() = there.resume(this, this)
  Val return(Val v) = v
  Val lookup(String x)
    = with (new DefaultRaise<Val>(new Nil())) {
        this.map.get(x)
      }
  Env extend(String x, Val v)
    = new Env(this.map.put(x, v))
}

class Store(Int id = 0,
            Map<Val,Val> map = new Map<Val,Val>())
    implements Handler<Tuple<Store, Val>, Val> {

  eff Val get(Val c)
    = there.resume(
        with (new DefaultRaise<Val>(new Nil())) {
          this.map.get(c)
        },
        this)
  eff Val put(Val c, Val v)
    = there.resume(v,
        new Store(this.id, this.map.put(c, v)))
  eff Val alloc() = {
    Cell c = new Cell(this.id);
    there.resume(c,
      new Store(this.id + 1,
        this.map.put(c, new Nil())))
  }
  Tuple<Store, Val> return(Val v)
    = new Tuple<Store,Val>(this, v)
}
```

References

[1] Jonathan Aldrich. 2013. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software (Onward! 2013)*. ACM, 101–116.

[2] D. Ancona, G. Lagorio, and E. Zucca. 2001. A Core Calculus for Java Exceptions. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 16–30. https://doi.org/10.1145/504282.504284

[3] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/2628136.2628149

[4] 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. https://doi.org/10.1016/j.jlamp.2014.02.001 Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

[5] Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: Extensible Algebraic Effects in Scala (Short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (SCALA 2017)*. ACM, New York, NY, USA, 67–72. https://doi.org/10.1145/3136000.3136007

[6] William R Cook. 2009. On understanding data abstraction, revisited. *ACM SIGPLAN Notices* 44, 10 (2009), 557–572.

[7] William R. Cook. 2012. A Proposal for Simplified, Modern Definitions of "Object" and "Object Oriented". (November 2012). Online: https://wcook.blogspot.nl/2012/07/proposal-for-simplified-modern.html.

[8] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

[9] Pablo Inostroza and Tijs van der Storm. 2017. Modular interpreters with implicit context propagation. *Computer Languages, Systems & Structures* 48 (2017), 39 – 67. https://doi.org/10.1016/j.cl.2016.08.001 Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE'15).

[10] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 145–158. https://doi.org/10.1145/2500365.2500590

[11] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

[12] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

[13] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/2103656.2103691

[14] Daan Leijen. 2017. Type Directed Compilation of Row-typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

[15] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

[16] Daniel Marino and Todd Millstein. 2009. A Generic Type-and-effect System. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1481861.1481868

[17] Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94.

[18] Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[19] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 258–282. https://doi.org/10.1007/978-3-642-31057-7_13

[20] Matías Toro and Éric Tanter. 2015. Customizable Gradual Polymorphic Effects for Scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 935–953. https://doi.org/10.1145/2814270.2814315

[21] Marko van Dooren and Eric Steegmans. 2005. Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions Using Anchored Exception Declarations. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 455–471. https://doi.org/10.1145/1094811.1094847

[22] Philip Wadler. 1998. The Expression Problem. Online. http://www.daimi.au.dk/~madst/tool/papers/expression.txt.

[23] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38 – 94. https://doi.org/10.1006/inco.1994.1093

[24] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2633357.2633358