An Effect System for
Algebraic Effects and Handlers

Andrej Bauer
andrej@andrej.com

Matija Pretnar
matija@pretnar.info

Department of Mathematics and Physics
University of Ljubljana, Slovenia

Abstract
We present an effect system for algebraic effects and handlers. Because handlers may transform an effectful computation into a pure one, the effect system is non-monotone in the sense that effects do not just accumulate, but may also be deleted from types or generally transformed. We also provide denotational semantics for the effect system, based on a domain-theoretic model with partial equivalence relations. The semantics validates equational reasoning about effectful computations.

1 Introduction

An effect system supplements a traditional type system for a programming language with information about which computational effects may, will, or will not happen when a piece of code is executed. A well designed and solidly implemented effect system helps the programmer understand the code and find mistakes, but it can also be used to safely rearrange, optimize, and parallelize code [10, 6]. As many before us [10, 20, 21, 5] we take on the task of striking just the right balance between simplicity and expressivity by devising an effect system for the programming language Eff [1]. The novelty here is that Eff has not only first-class algebraic effects [15], but also effect handlers [16]. Therefore, an effect system for Eff is by its nature non-monotone — an effectful computation may become pure when enclosed by a handler — so effects do not just accumulate in the types, but also get deleted and generally transformed. Another feature of our effect system is that its denotational semantics validates equational reasoning, which is traditionally thought to be in the dominion of pure languages, and can be tricky once effects are included [7].

The paper is organized as follows. In §2 we describe the core Eff and an effect system for it. In §3 we give a denotational semantics for core Eff, and use it to validate program transformation rules that can be used for equational reasoning about effectful computations.
2 Core Eff

Eff is a ML-style [11, 8] programming language. Effects in ML are not visible in the types. For example, inserting a print statement in the middle of code does not create any changes in the typing information. In contrast, in the monadic style such a change would taint all enclosing types with the I/O monad [2]. It is our intention to augment the types with information about computational effects which is unobtrusive for programmers, i.e., in the implementation we envision effect inference which serves primarily as a program analysis tool.

In ML-style languages effects are provided through built-in functions and special-purpose constructs such as exceptions and references. Eff is based on the algebraic approach in which effects are accessed uniformly and exclusively through operations, which are a primitive concept. Examples of operations are reading and writing on a communication channel, updating and looking up the contents of a reference, and raising an exception. Thus, in Eff each terminating computation results either in an (effect-free) value, or it triggers an operation. Each operation has an associated (delimited) continuation, which is a suspended computation expecting the result of the operation and doing whatever is to be done after the operation is performed.

Operations by themselves do not actually perform effects. Instead their behavior is controlled by a second primitive notion, the effect handlers. These are a direct generalization of exception handlers to other operations. The most significant difference between exception handlers and effect handlers is that the latter have access to the continuation of the handled operation. With handlers we may implement a great variety of computational effects, such as transactional memory, various non-deterministic execution strategies, stream redirection, cooperative multi-threading, delimited continuations, and others.

The current implementation of Eff includes a number of features, such as syntactic sugar, products, records, inductive types, type definitions, effect definitions, etc., which are inessential for a conceptual analysis. We therefore restrict attention to core Eff whose syntax is shown below.

We describe informally what the various parts mean, and refer the readers to [1] for a more thorough introduction. Eff uses a fine grain call-by-value evaluation strategy [9], which means that it distinguishes effect-free expressions and possibly effectful computations. There are corresponding expression and computation types.

2.1 Expressions

An expression is either a variable \( x \), zero \( 0 \), a successor \( \text{succ\ } e \), a boolean value \( \text{true} \) or \( \text{false} \), the unit \( () \), an effect instance \( \iota \), a function abstraction of a computation, an operation \( e \# \text{op} \), or a handler transforming computations to computations. We briefly comment on the ones that are peculiar to Eff.

An effect type

\[
\text{effect (operation } \text{op}_i: A_i \rightarrow B_i)_i \text{ end}
\]

declares operations \( \text{op}_i \) with given types of parameters \( A_i \) and results \( B_i \). (Here and elsewhere, \( (\cdots)_i \) indicates that \( \cdots \) may be repeated finitely many times.) For example,
Types

Effect type \( E ::= \text{effect} \ (\text{operation} \ op : A_i \to B_i), \end \)

Expression type \( A, B ::= \text{nat} \mid \text{bool} \mid \text{unit} \mid \text{empty} \mid E^{(i_1, \ldots, i_n)} \mid A \to C \mid C \Rightarrow D \)

Computation type \( C, D ::= A!^{(i_1 \# op_1, \ldots, i_n \# op_n)} \)

Terms

Expression \( e ::= x \mid 0 \mid \text{succ} \ e \mid \text{true} \mid \text{false} \mid () \mid \ell \mid \text{fun} \ x \mapsto c \mid e \# op \mid h \)

Handler \( h ::= \text{handler} \ \text{val} \ x \mapsto c_v \mid (e_i \# op_i x_i \mapsto c_i)_i \)

Computation \( c ::= \text{val} \ e \mid \text{let} \ x = c_1 \text{ in } c_2 \mid \text{let rec} \ f x = c_1 \text{ in } c_2 \mid \text{iszero} \ e \mid \text{if} \ e \text{ then } c_1 \text{ else } c_2 \mid \text{absurd} \ e \mid e_1 e_2 \mid \text{with } e \text{ handle } c \)

the effect type of exceptions is

\[
\text{effect} \\
\text{operation abort : unit} \to \text{empty} \\
\text{end}
\]

and the effect type \( \text{ref} \) for a reference holding a natural number is

\[
\text{effect} \\
\text{operation lookup : unit} \to \text{nat} \\
\text{operation update : nat} \to \text{unit}
\]

(1)

Effect instances \( i \) are a way of making several copies of the same computational effect. For example, there may be several communication channels, several mutable references, etc. In this respect Eff differs from [5], where bare operations are considered; in terms of Eff that is like having a single instance of each effect.

The expression type \( E^{(i_1, \ldots, i_n)} \) is inhabited by expressions which evaluate to one of the instances \( i_1, \ldots, i_n \), whose effect type is \( E \). We call \( \{i_1, \ldots, i_n\} \) a region and abbreviate it as \( \rho \). A smaller region is more informative, so ideally we would like all of them to be just singletons. But this is not possible because instances are first-class values and so we can write

\[
\text{let } x = (\text{if } b \text{ then } i_1 \text{ else } i_2) \text{ in } \ldots
\]

The best we can say about \( x \) is that its type is \( E^{(i_1, i_2)} \).
2.2 Computations

A computation type $A\{\iota_1 \# op_1, \ldots, \iota_n \# op_n\}$ means that the computation either produces a (pure) value of type $A$, or triggers one of the listed operations $\iota_1 \# op_1, \ldots, \iota_n \# op_n$. We abbreviate such finite sets of operations with the letter $\delta$ and call them dirt. A computation is either a pure value $val$, a let binding, a recursive function definition, a zero-test $\text{iszero } e$, a conditional statement, a destructor for the empty type, an application, or a handle construct.

The computation $val$ is pure and indicates a “final” result $e$, while an operation applied to a parameter $\iota \# op e$ is the principal way of triggering an effect. By itself $\iota \# op e$ is just a suspended computation with an associated continuation. For an actual effect to take place it has to be handled by a handler, as described below. The continuation associated with $\iota \# op e$ is $fun x \mapsto \text{val } x$. Such operations are known as general effects [15].

A binding $\text{let } x = c_1 \text{ in } c_2$ is evaluated as follows:

1. if $c_1$ evaluates to $val e$ then the binding evaluates to $c_2$ with $x$ bound to $e$,
2. if $c_1$ evaluates to an operation $\iota \# op e$ with continuation $\kappa$, then the binding evaluates to $\iota \# op e$ with continuation $fun y \mapsto (\text{let } x = \kappa y \text{ in } c_2)$.

It may help to think of $val$ and $let$ as being similar to Haskell $\text{return}$ and $\text{do}$, respectively. In ML $val$ is invisible, while $let$ is essentially the same as ours.

The handling construct applies a handler to a computation. If $h$ is the handler

$$\text{handler } val \ x \mapsto c_v | (\iota_i \# op_i x_i k_i \mapsto c_i)_{i}$$

and $c$ is a computation then with $h \text{ handle } c$ first evaluates $c$ and then evaluates the clause of the handler that matches the result given by $c$. If no clause matches, $c$ evaluated to an operation which is propagated outwards. In all cases the handler wraps itself around the continuation so that subsequent operations are handled as well. More precisely:

1. if $c$ evaluates to $val e$, then the handling construct evaluates to $c_v$ with $x$ bound to $e$,
2. if $c$ evaluates to $\iota_i \# op_i e'$ with continuation $\kappa$, then the handling construct evaluates to $c_i$ with $x_i$ and $k_i$ bound to $e'$ and $fun y \mapsto with \ h \ handle \ \kappa y$, respectively.
3. if $c$ evaluates to any other operation $\iota \# op e'$ with continuation $\kappa$, then the handling construct acts as if $h$ contained the clause

$$\iota \# op x k \mapsto (\text{let } y = \iota \# op x \text{ in } k y).$$

Thus it evaluates to $\iota \# op e'$ with continuation $fun y \mapsto with \ h \ handle \ (\kappa y)$.

We may wrap several handling constructs around $c$, in which case the inner handler takes precedence. Note that $\text{let } x = c_1 \text{ in } c_2$ is equivalent to

$$\text{with } (\text{handler } val \ x \mapsto c_2) \text{ handle } c_1$$

so we could theoretically omit $\text{let}$. 
2.3 Typing rules

The typing system of core Eff has two typing judgments,

\[ \Gamma \vdash e : A \quad \text{and} \quad \Gamma \vdash c : C \]

stating that an expression \( e \) has expression type \( A \), and that a computation \( c \) has computation type \( C \), respectively. As usual, \( \Gamma \) is a typing context

\[ x_1 : A_1, \ldots, x_n : A_n \]

which assigns expression types \( A_i \) to distinct variables \( x_i \). We also have subtyping of expression types \( A \leq A' \) and computation types \( C \leq C' \). We fix an assignment \( \Xi \) of effect types to instances, and assume implicitly that all dirt and regions appearing in the rules are well-formed, i.e., if \( \iota \# \text{op} \) appears then in fact \((\iota : E) \in \Xi \) and \( \text{op} \in E \), and that a region \( \rho \) in \( E^\rho \) mentions only instances whose effect type is \( E \).

The typing rules for variables, primitive constants and functions are standard:

\[
\begin{align*}
(x : A) \in \Gamma & \quad \Gamma \vdash x : A \\
0 : \text{nat} & \quad \Gamma \vdash 0 : \text{nat} \\
\text{suc} e : \text{nat} & \quad \Gamma \vdash \text{suc} e : \text{nat} \\
\emptyset : \text{unit} & \quad \Gamma \vdash \emptyset : \text{unit} \\
\text{true} : \text{bool} & \quad \Gamma \vdash \text{true} : \text{bool} \\
\text{false} : \text{bool} & \quad \Gamma \vdash \text{false} : \text{bool} \\
\text{fun} x \mapsto c & \quad \Gamma, x : A \vdash c : C \\
\text{handler value} x \mapsto (e_i \# \text{op}_i x_i \mapsto c_i) & \quad \Gamma \vdash (\text{handler value} x \mapsto e_i \# \text{op}_i x_i \mapsto c_i) : A!\delta \Rightarrow C
\end{align*}
\]

The typing rule for instances

\( \iota \in \rho \)

verifies that the instance \( \iota \) is in the region \( \rho \) and consults \( \Xi \) to check that \( \rho \) contains only instances whose effect type is \( E \). An operation \( e \# \text{op} \) has a function type, where the dirt in the codomain must contain operations \( \iota \# \text{op} \) for \( \iota \) ranging over the region associated with \( e \):

\[
\begin{align*}
\Gamma \vdash e : E^\rho \\
(\text{op} : A \rightarrow B) \in E \\
\forall \iota \in \rho, (\iota \# \text{op}) \in \delta \\
\Gamma \vdash e \# \text{op} : A \Rightarrow B!\delta
\end{align*}
\]

The handler type \( A!\delta \Rightarrow C \) expresses the fact that a handler transforms computations of type \( A!\delta \) to computations of type \( C \). The typing rule for handlers

\[
\begin{align*}
\Gamma, x : A \vdash c_v : C \\
\text{op}_i : A_i \rightarrow B_i \in E_i \\
\forall (\iota \# \text{op}) \in \delta, \exists i, \rho_i = \{\iota\} \land \text{op}_i = \text{op} \\
\Gamma \vdash (\text{handler value} x \mapsto e_v \mid (e_i \# \text{op}_i x_i \mapsto c_i)_i) : A!\delta \Rightarrow C
\end{align*}
\]

checks that the instances and operations are paired up correctly according to their effect types, all clauses have computation type \( C \) under suitable assumptions on bound variables, and every operation in \( \delta \) is handled by a clause. Note that an operation \( \iota \# \text{op} \)
is taken to be handled by a clause $c_i \# \text{op}_i \ x_i \ k_i \mapsto c_i$ only if \text{op} = \text{op}_i and $c_i$ is ascertained to have exactly the region $\{t\}$. If the region were $\{t, t'\}$ we could not tell whether the clause handles $t \# \text{op}$ or $t' \# \text{op}$.

The typing rules for computations hold no surprises at all:

$$
\Gamma \vdash e : A \\
\Gamma \vdash \text{val} \ e : A!\delta \\
\Gamma \vdash (\text{let } x = c_1 \ \text{in } c_2) : B!\delta
$$

$$
\Gamma, f : A \rightarrow C \ x : A \vdash e_1 : C \\
\Gamma, f : A \vdash e_2 : D \\
\Gamma \vdash (\text{let } \text{rec} \ f \ x = c_1 \ \text{in } c_2) : D
$$

$$
\Gamma \vdash e : \text{nat} \\
\Gamma \vdash e : \text{bool} \\
\Gamma \vdash (\text{if } e \ \text{then } c_1 \ \text{else } c_2) : C
$$

$$
\Gamma \vdash e : \text{empty} \\
\Gamma \vdash (\text{absurd } e) : C
$$

The structural subtyping rules are expected as well [3]:

$$
\Gamma \vdash e : A \quad A \leq A' \\
\Gamma \vdash e : A' \\
\Gamma \vdash e : C \quad C \leq C' \\
\Gamma \vdash e : C'
$$

$$
A \leq A' \\
A' \leq A'' \\
A \leq A''
$$

$$
A' \leq A \\
C \leq C' \\
C' \leq D \\
(C \Rightarrow D) \leq (C' \Rightarrow D')
$$

$$
\rho \subseteq \rho' \\
E^\rho \leq E'^\rho
$$

$$
A \leq A' \\
\delta \subseteq \delta' \\
A!\delta \leq A'!\delta'
$$

There are two subsumption rules, one for each typing judgment. The subtyping relation is reflexive and transitive. Function and handler types are contravariant in the domain and covariant in the codomain. Indeed, we may always pretend that a handler handles fewer operations, and triggers more operations than it actually does. An effect type is covariant in the region, while a computation type is covariant in both its parts.

Subtyping buys us additional expressivity. For instance, assuming $\iota_1$ and $\iota_2$ are instances of $E$,

```latex
\text{let } a = \iota_1 \text{ in} \\
\text{let } b = (\text{if } p \ \text{then } a \ \text{else } \iota_2) \text{ in} \\
\text{val} (\text{handler val } x \mapsto \text{val } \emptyset | a \# \text{op}_x \ k \mapsto \text{val } \emptyset)
```

we would like to give $a$ the type $E^{(\iota_1)}$ so that the handler can have the more specific type $A!\{\iota_1 \# \text{op}\} \Rightarrow \text{unit}\emptyset$, but then without subtyping $E^{(\iota_1)} \leq E^{(\iota_1, \iota_2)}$ we could not give $b$ its type $E^{(\iota_1, \iota_2)}$. 

6
2.4 Example: state handlers

We demonstrate the features of our type system by looking at a state handler for the reference effect \( \text{ref} \) shown in (1). Define \( \text{state} \) to be the function

\[
\begin{align*}
\text{fun } r & \mapsto (\text{handler } \text{val } x \mapsto \text{val } (\text{fun } s \mapsto \text{val } x)) \\
| r \# \text{lookup } x k & \mapsto \text{val } (\text{fun } s \mapsto k s s) \\
| r \# \text{update } s' k & \mapsto \text{val } (\text{fun } s \mapsto k (s' s))
\end{align*}
\]

In words, \( \text{state} \) accepts an instance \( r \) and returns a handler which handles lookups and updates on \( r \) by using the standard functional encoding of the state monad. For any instance \( \iota \), expression type \( A \) and dirt \( \delta \), the function \( \text{state} \) has the type

\[
\text{ref}\{\iota\} \to (A!(\{\iota \# \text{lookup}, \iota \# \text{update}\} \cup \delta) \Rightarrow (\text{nat} \to A!\delta)!\delta).
\]

Thus, if \( c \) is a computation of type

\[
A!(\{\iota \# \text{lookup}, \iota \# \text{update}\} \cup \delta)
\]

then

\[
\text{let } h = \text{state } \iota \text{ in (with } h \text{ handle } c) \quad (2)
\]

has the type \((\text{nat} \to A!\delta)!\delta\). That is, we obtained a computation which possibly triggers effects \( \delta \) and returns a function. Upon application of the function to an initial state effects \( \delta \) may be triggered again, after which a final result of type \( A \) is obtained. In particular, if \( \delta = \emptyset \) then (2) is a pure computation.

If we weakened the domain of \( \text{state} \) to \( \text{ref}\{\iota_1, \iota_2\} \) then we would not be able to deduce that \( \text{state} \) handled anything, so we would only be able to assign to \( \text{state} \) the less useful type

\[
\text{ref}\{\iota_1, \iota_2\} \to (A!\delta \Rightarrow (\text{nat} \to A!\delta)!\delta).
\]

We may handle several references by invoking several instantiations of \( \text{state} \). For example, let \( c \) be the computation which swaps the contents of two references:

\[
\begin{align*}
\text{let } x_1 &= \iota_1 \# \text{lookup}() \text{ in} \\
\text{let } x_2 &= \iota_2 \# \text{lookup}() \text{ in} \\
\text{let } u &= \iota_1 \# \text{update } x_2 \text{ in} \\
\text{let } v &= \iota_2 \# \text{update } x_1 \text{ in }
\end{align*}
\]

By itself, \( c \) has the type

\[
\text{unit}!(\{\iota_1 \# \text{lookup}, \iota_1 \# \text{update}, \iota_2 \# \text{lookup}, \iota_2 \# \text{update}\},
\]

while

\[
\text{let } h_1 = \text{state } \iota_1 \text{ in (with } h_1 \text{ handle } c)
\]

has type \((\text{nat} \to \text{unit}!(\{\iota_2 \# \text{lookup}, \iota_2 \# \text{update}\})!(\iota_2 \# \text{lookup}, \iota_2 \# \text{update}))!\{\iota_2 \# \text{lookup}, \iota_2 \# \text{update}\}\). If we handle both instances,

\[
\begin{align*}
\text{let } h_1 &= \text{state } \iota_1 \text{ in} \\
\text{let } h_2 &= \text{state } \iota_2 \text{ in} \\
\text{with } h_2 \text{ handle (with } h_1 \text{ handle } c),
\end{align*}
\]
we get the pure type \( \text{nat} \rightarrow (\text{nat} \rightarrow \text{unit!}\emptyset)!\emptyset \).

### 3 Denotational semantics

An outline of a set-theoretic algebraic semantics of computational effects, as developed by \([1, 4, 14, 15, 16]\), is shown in the following table.

| Programming language | Algebra |
|----------------------|---------|
| effect type          | algebraic signature |
| expression type      | set |
| expression           | element |
| computation type     | free algebra |
| pure computation     | generator |
| effectful computation| algebraic operation |
| handler              | homomorphism of algebras |

To properly account for recursion and non-termination we adapt it to a domain-theoretic semantics of algebraic effects under which expression and computation types are domains. We first give Curry-style semantics in which terms are interpreted without being typed, and then, following John Reynolds \([18]\), we provide a Church-style semantics in which types receive meanings as well. It does not matter much what kind of domains we use, they could be \(\omega\)-cpos, Scott domains, effective Scott domains, or any other kind of domains that model the basic type-theoretic operations (product, function space, coalesced sum).

#### 3.1 Semantics of expressions and computations

Expressions could be modeled with predomains, because they are inert pieces of data, free from computational effects, including non-termination. However, the bottom element is useful for denotation of ill-typed expressions and runtime errors. In any case, the predomain nature of expressions will be captured later on by the partial equivalence relations. The domains for computation types are free in a suitable sense, i.e., they enjoy a recursion principle which acts as a substitute for the universality of free algebras. We assume a given set of all instances \(\mathbb{I}\) and a set of all operation symbols \(\mathbb{O}\), write \(+\) for a disjoint sum, and \(\oplus\) for coalesced sum.

To interpret expressions and computations we need suitable domains of values \(V\) and results \(R\), respectively. These have to be large enough to contain all possible denotations of expressions and computations, which is usually achieved by solving suitable recursive domain equations, such as

\[
V = \mathbb{N}_\bot \oplus \{0, 1\}_\bot \oplus \{\star\}_\bot \oplus \mathbb{I}_\bot \oplus R^V \oplus R^R,
\]

\[
R = (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)\_\bot.
\]

The summands in the equation for \(V\) correspond to various expression types. The recursive domain equation for \(R\) says that a non-bottom element of \(R\) is either a value,
or a quadruple \((\iota, \text{op}, v, \kappa)\) corresponding to the operation \(\iota \# \text{op}\) applied to parameter \(v\) and with continuation \(\kappa\). However, as in [18], we shall assume only that \(V\) and \(R\) are large enough to contain the various components needed for the semantics as retracts:

\[
\begin{align*}
N &\xrightarrow{\text{sval}} V &\quad \{0, 1\} &\xrightarrow{\text{stoi}} V &\quad \{\ast\} &\xrightarrow{\text{stoi}} V \\
\perp &\xrightarrow{\text{reflect}} V &\quad R^V &\xrightarrow{\text{refl}} V &\quad R^R &\xrightarrow{\text{refl}} V
\end{align*}
\]

and

\[
(V + \perp \times \perp \times V \times R^V) \xrightarrow{\text{refl}} R
\]

Thus, \(R\) and \(V\) could be solutions to the above domain equations, but they could also both be a universal domain, or just a (non-trivial) reflexive domain. Note that there are further canonical retractions

\[
\begin{align*}
V &\xrightarrow{\text{sval}} (V + \perp \times \perp \times V \times R^V) \\
(\perp \times \perp \times V \times R^V) &\xrightarrow{\text{oper}} (V + \perp \times \perp \times V \times R^V)
\end{align*}
\]

which in combination with the section-retraction pair \((s_{\text{res}}, r_{\text{res}})\) convert elements of \(R\) to pure values and operations, respectively.

The domain \((V + \perp \times \perp \times V \times R^V)\) has the following recursion principle. Given a domain \(D\) and maps

\[
h_{\text{val}} : V \to D \quad \text{and} \quad h_{\text{oper}} : \perp \times \perp \times V \times R^V \to D,
\]

there is a unique strict map \(h : (V + \perp \times \perp \times V \times R^V) \to D\) such that, for all \(v \in V, \iota \in I, \text{op} \in O, \kappa : V \to R,\)

\[
\begin{align*}
h(s_{\text{val}}(v)) &= h_{\text{val}}(v), \\
h(s_{\text{oper}}(\iota, \text{op}, v, \kappa)) &= h_{\text{oper}}(\iota, \text{op}, v, h \circ r_{\text{res}} \circ \kappa).
\end{align*}
\]

Moreover, \(h\) depends continuously on the data \(h_{\text{val}}\) and \(h_{\text{oper}}\). For example, given a map \(f : V \to R\), there is a unique strict map

\[
f^\dagger : (V + \perp \times \perp \times V \times R^V) \to R,
\]

called the lifting of \(f\), which depends on \(f\) continuously and satisfies the recursive equations

\[
f^\dagger(s_{\text{val}}(v)) = f(v),
\]

\[
f^\dagger(s_{\text{oper}}(\iota, \text{op}, v, \kappa)) = s_{\text{oper}}(\iota, \text{op}, v, f^\dagger \circ r_{\text{res}} \circ \kappa).
\]

\(^1\text{A domain } D \text{ is reflexive if it contains its functions space } D^D \text{ as a retract, and is therefore a model of the untyped } \lambda\text{-calculus.}\)
An environment $\eta$ is a map from variable names to values. We denote by $\eta[x \mapsto v]$ the environment which assigns $v$ to $x$ and otherwise behaves as $\eta$. The untyped denotational semantics of expressions assigns to each expression a map from environments to $V$, as follows. First we have the standard constructs:

- $[x] \eta = \eta(x)$
- $[\emptyset] \eta = \text{unit}(\ast)$
- $[0] \eta = \text{nat}(0)$
- $[\text{succ } e] \eta = \begin{cases} \text{nat}(n + 1) & \text{if } r_{\text{nat}}([e] \eta) = n \in \mathbb{N}, \\ \text{nat}(\bot) & \text{if } r_{\text{nat}}([e] \eta) = \bot \end{cases}$
- $[\text{true}] \eta = \text{bool}(1)$
- $[\text{false}] \eta = \text{bool}(0)$
- $[\lambda x. f] \eta = \text{fun}(\lambda v. ([f] (\eta[x \mapsto v])))$

When $e$ evaluates to an instance we interpret $e \# \text{op}$ as a generic effect:

$$[e \# \text{op}] \eta = \begin{cases} s_{\text{op}}(\lambda v. V \cdot s_{\text{res}}(\text{op}(i, \text{op}, v, V \circ s_{\text{val}}))) & \text{if } r_{\text{effect}}([e] \eta) = i \in \mathbb{I}, \\ s_{\text{op}}(\lambda v. V \cdot s_{\text{res}}(\bot)) & \text{if } r_{\text{effect}}([e] \eta) = \bot. \end{cases}$$

The interpretation of a handler is

$$[\text{handler } \text{val } x \mapsto e_v \mid (e_i \# \text{op}_i x, k_i \mapsto c_i)](\eta) = s_{\rightarrow}(h \circ r_{\text{res}})$$

where $h : (V + \mathbb{I} \times \emptyset \times V \times R^V)_{\bot} \rightarrow R$ is characterized using the recursion principle for $R$ as follows:

1. if one of the $r_{\text{effect}}([e_i] \eta)$ is $\bot$ we set $h = \lambda x. s_{\text{res}}(\bot)$, otherwise
2. if $r_{\text{effect}}([e_i] \eta) = i_i \in \mathbb{I}$ for all $i$, then we define $h$ by cases as

   $$h(s_{\text{val}}(v)) = [c_v] (\eta[x \mapsto v])$$
   $$h(s_{\text{op}}(i, \text{op}, v, \kappa)) = [c_i] (\eta[x_i \mapsto v, k_i \mapsto h \circ r_{\text{res}} \circ \kappa])$$
   for all $i$,
   $$h(s_{\text{op}}(i, \text{op}, v, \kappa)) = s_{\text{res}}(s_{\text{op}}(i, \text{op}, v, h \circ r_{\text{res}} \circ \kappa))$$
   if $(i, \text{op}) \neq (i_i, \text{op}_i)$ for all $i$.

There may be overlapping cases in the second clause, i.e., it could happen that $(i_i, \text{op}_i) = (i_j, \text{op}_j)$, in which case the clause that is listed first counts.

Computations are modeled as maps from environments to results. Promotion of expressions to computations is just the inclusion of $V$ into $R$,

$$[\text{val } e] \eta = s_{\text{res}}(s_{\text{val}}([e] \eta)).$$
The \texttt{let} statement corresponds to monadic-style binding:

\[
\texttt{let } x = c_1 \texttt{ in } c_2 \eta = (\lambda v : V. [c_2] (\eta[x \mapsto v])) \circ (r_{\text{res}}([c_1] \eta)),
\]

A recursive function definition is interpreted as

\[
\texttt{let rec } f x = c_1 \texttt{ in } c_2 \eta = [c_2] (\eta[f \mapsto s \mapsto (t)])
\]

where \( t : V \rightarrow R \) is the least fixed point of the map

\[
t \mapsto s \mapsto (\lambda v : V. [c_1] \eta[x \mapsto \text{val}])
\]

The elimination forms are interpreted in the usual way as:

\[
\begin{align*}
\texttt{iszero} \; e \; \eta &= \begin{cases} 
    s_{\text{res}}(s_{\text{val}}(s_{\text{bool}}(1))) & \text{if } 0 = r_{\text{nat}}([e] \eta) \\
    s_{\text{res}}(s_{\text{val}}(s_{\text{bool}}(0))) & \text{if } 0 \neq r_{\text{nat}}([e] \eta) \in \mathbb{N} \\
    s_{\text{res}}(\bot) & \text{if } \bot = r_{\text{nat}}([e] \eta)
  \end{cases} \\
\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \; \eta &= \begin{cases} 
    [c_1] \eta & \text{if } r_{\text{bool}}([e] \eta) = 1 \\
    [c_2] \eta & \text{if } r_{\text{bool}}([e] \eta) = 0 \\
    s_{\text{res}}(\bot) & \text{otherwise}
  \end{cases} \\
\texttt{absurd} \; e \; \eta &= s_{\text{res}}(\bot) \\
\end{align*}
\]

Finally, the handling construct is just an application

\[
\texttt{with } e \texttt{ handle } c \; \eta = r_{\mapsto}([e] \eta)([c] \eta).
\]

The denotational semantics of expressions and computations is written in such a way that it immediately suggests a big-step operational semantics. Indeed, in the implementation of Eff the main evaluation function is essentially a transcription of the above rules into OCaml [8].

### 3.2 Semantics of types

According to John Reynolds, the untyped semantics of terms given in \[3.1\] is related to a semantics of types by specifying, for all expression types \( A \) and computation types \( C \), section-retraction pairs

\[
[A] \xrightarrow{t_A} V \quad \text{and} \quad [C] \xrightarrow{t_C} R,
\]

A value \( v \in V \) and a result \( r \in R \) have types \( A \) and \( C \), respectively, if they are members of the corresponding retracts, which is expressed by fix-point equations

\[
v = s_A(r_A(v)) \quad \text{and} \quad r = s_C(r_C(r)).
\]

Such semantics works well for the \textit{structural} part of types, i.e., everything except the effect system. Therefore, for the effect system we are going to use a second level of
semantics by equipping the retracts with *partial equivalence relations (per)* \[^2\] This way the semantics is neatly stratified: we may ignore the effect system and use only the domain-theoretic part of semantics to get a more traditional account of the language, or we also include the effect system and the pers for a more refined meaning of types.

For the basic types, the effect types and the function type we set

\[
\begin{align*}
\llbracket \text{nat} \rrbracket &= \mathbb{N}_\perp, \\
\llbracket \text{unit} \rrbracket &= \{\ast\}_\perp, \\
\llbracket E^\rho \rrbracket &= I_\perp, \\
\llbracket \text{bool} \rrbracket &= \{0, 1\}_\perp, \\
\llbracket \emptyset \rrbracket &= \emptyset_\perp, \\
\llbracket A \to C \rrbracket &= \llbracket C \rrbracket^{[A]},
\end{align*}
\]

The section-retraction pairs for the basic types and the effect types are the ones given previously, and for the function type we set

\[
\begin{align*}
s_{A \to C}(f) &= s_A \circ f \circ r_A \\
r_{A \to C}(v) &= r_C \circ r_{\text{type}}(v) \circ s_A.
\end{align*}
\]

The pers for the basic types nat, bool, unit, and empty are identities on the *total* elements, while the per for an effect type \(E^\rho\) is identity restricted to \(\rho\):

\[
\begin{align*}
v \sim_{\text{nat}} v' &\iff v = v' \in \mathbb{N}, \\
v \sim_{\text{bool}} v' &\iff v = v' \in \{0, 1\}, \\
v \sim_{\text{unit}} v' &\iff v = v' = \ast, \\
v \sim_{\text{empty}} v' &\iff v \in \rho.
\end{align*}
\]

The function type \(A \to C\) has the standard per

\[
f \sim_{A \to C} f' \iff \forall v, v' \in \llbracket A \rrbracket. (v \sim_{A} v' \implies f(v) \sim_{C} f'(v')).
\]

Similarly, handler types \(C \Rightarrow D\) are treated as functions from computations to computations. The underlying domain is the function space domain with the section-retraction pair

\[
\llbracket C \Rightarrow D \rrbracket = \llbracket D \rrbracket^{[C]} \quad \text{and} \quad \llbracket C \Rightarrow D \rrbracket \xrightarrow{s_{C \Rightarrow D}} V,
\]

defined in the expected way,

\[
\begin{align*}
s_{C \Rightarrow D}(h) &= s_D \circ s_C \circ h \circ r_C, \\
r_{C \Rightarrow D}(v) &= r_C \circ r_{\text{type}}(v) \circ s_C.
\end{align*}
\]

Also the per is the one for function types,

\[
h \sim_{C \Rightarrow D} h' \iff \forall r, r' \in \llbracket C \rrbracket. (r \sim_{C} r' \implies h(r) \sim_{D} h'(r')).
\]

Semantics of computation types is a bit more involved. Let \(\mathbb{D}\) be the set of all pairs \((\iota, \text{op})\) where \(\iota\) is an instance whose associated effect type is \(E\) and \(\text{op} : A_{\text{op}} \to B_{\text{op}}\) is an operation declared by \(E\). The domain associated with the computation type \(A!\delta\) is the solution of recursive domain equation

\[
\llbracket A!\delta \rrbracket = (\llbracket A \rrbracket + \coprod_{(\iota, \text{op}) \in \mathbb{D}} [A_{\text{op}}] \times [A!\delta][B_{\text{op}}])_\perp.
\]

\[^2\]Recall that a per is a symmetric and transitive relation.
Because we want to keep the domain-theoretic part of semantics independent of the effect system, we do not use the information provided by \( \delta \), and so we have to sum over all operations in \( \mathbb{D} \). We encode information about \( \delta \) in the per \( \sim_{\text{As}} \). There are canonical retractions

\[
\begin{align*}
\llbracket A \rrbracket & \xrightarrow{\sim_{\text{As}}} \llbracket \text{As} \rrbracket \quad \text{and} \quad (\bigoplus_{(i, \text{op}) \in \mathbb{D}} \llbracket A_{\text{op}} \rrbracket \times \llbracket \text{As} \rrbracket)^\perp \xrightarrow{\sim_{\text{As}}} \llbracket \text{As} \rrbracket
\end{align*}
\]

Just like for \( R \), we shall not use the domain equation for \( \llbracket \text{As} \rrbracket \), but rather only the above section-retraction pairs. The retraction \( r_{\text{As}} : \llbracket \text{As} \rrbracket \rightarrow \llbracket \text{As} \rrbracket \) is the composition of \( r_{\text{res}} \) and the map \( f : (V + \mathbb{I} \times \mathbb{C} \times V \times R') \rightarrow \llbracket \text{As} \rrbracket \) defined by the recursion principle as

\[
\begin{align*}
f(s_{\text{val}}(v)) &= s^{\text{As}}_{\text{val}}(r_A(v)), \\
f(s_{\text{oper}}(i, \text{op}, v, \kappa)) &= \begin{cases} s^{\text{As}}_{\text{oper}}(i, \text{op}, r_{A_{\text{op}}}(v), f \circ r_{\text{res}} \circ \kappa \circ s_{B_{\text{op}}}) & \text{for } (i, \text{op}) \in \mathbb{D}, \\
\perp & \text{for } (i, \text{op}) \notin \mathbb{D}. \end{cases}
\end{align*}
\]

The section \( s^{\text{As}} \) is defined similarly, using an analogous recursion principle for

\[
(\llbracket A \rrbracket + \bigoplus_{(i, \text{op}) \in \mathbb{D}} \llbracket A_{\text{op}} \rrbracket \times \llbracket \text{As} \rrbracket)^\perp.
\]

The per \( \sim_{\text{As}} \) is defined inductively as the least one satisfying:

1. \( \perp \sim_{\text{As}} \perp \),
2. for all \( v, v' \in \llbracket A \rrbracket \), if \( v \sim_A v' \) then \( s^{\text{As}}_{\text{val}}(v) \sim_{\text{As}} s^{\text{As}}_{\text{val}}(v') \),
3. for all \( (i \neq \text{op}) \in \delta \), all \( v, v' \in \llbracket A_{\text{op}} \rrbracket \), and all \( \kappa, \kappa' \in \llbracket B_{\text{op}} \rightarrow \text{As} \rrbracket \),

\[
v \sim_{A_{\text{op}}} v' \land \kappa \sim_{B_{\text{op}}} r_{A_{\text{op}}} \Rightarrow s^{\text{As}}_{\text{oper}}(i, \text{op}, v, \kappa) \sim_{\text{As}} s^{\text{As}}_{\text{oper}}(i, \text{op}, v', \kappa').
\]

The pers \( \sim_A \) and \( \approx_C \) are defined on the corresponding domains \( \llbracket A \rrbracket \) and \( \llbracket C \rrbracket \). We can transfer them along sections to pers \( \approx_A \) and \( \approx_C \) on \( V \) and \( R \) by

\[
v \sim_A v' \iff s_A(v) \approx_A s_A(v') \quad \text{and} \quad r \sim_C r' \iff s_C(r) \approx_C s_C(r').
\]

Note that \( v \approx_A v \) implies \( s_A(r_A(v)) = v \), and similarly for \( \approx_C \).

We connect the untyped semantics of terms and the semantics of types with a soundness theorem.

**Theorem 3.1 (Matching semantics of terms and types)** Let \( \Gamma \) be a typing context and \( \eta \) an environment such that, for every \( x_i : A_i \) in \( \Gamma \), \( \eta(x_i) \approx_A, \eta(x_i) \).

1. If \( \Gamma \vdash e : A \) then \( \llbracket e \rrbracket \eta \approx_A \llbracket e \rrbracket \eta \).
2. If \( \Gamma \vdash c : C \) then \( \llbracket c \rrbracket \eta \approx_C \llbracket c \rrbracket \eta \).
3. If $A \leq A'$ then $[[A]] = [[A']]$ and $(\sim A) \subseteq (\sim A')$.

4. If $C \leq C'$ then $[[C]] = [[C']]$ and $(\sim C) \subseteq (\sim C')$.

**Proof.** The proof proceeds by induction on the judgment derivations. The typing rule for recursive function definitions works because all the pers for computation types are admissible, i.e., they contain the least element and are closed under suprema of chains. Likewise, the rule for elimination of the empty typeworks because the pers for computation types contain the least element.

### 3.3 Equational reasoning

We can use the denotational semantics to validate program transformations. This is all very familiar, so we just review the main idea. Consider computations $c_1$ and $c_2$ which have type $C$ in the typing context $\Gamma$. Say that $c_1$ and $c_2$ are (semantically) equivalent and write $c_1 \equiv c_2$ when, for any environment $\eta$ such that $\eta(x_i) \approx A_i \eta(x_i)$ for all $(x_i : A_i) \in \Gamma$, we have $[[c_1]] \eta \approx [[c_2]] \eta$.

A similar definition can be made for equivalence of expressions. Then $\equiv$ is an equivalence relation which satisfies the “substitution of equals” principle, i.e., for a well-typed evaluation context $C\ [\ - \ ]$, $c_1 \equiv c_2$ implies $C[c_1] \equiv C[c_2]$, and similarly for expressions. Therefore, we may safely replace a computation or an expression with an equivalent one. In fact, since evaluation is a form of program transformation, we have a criterion for correctness of implementation: an evaluation strategy is correct with respect to the semantics if it preserves semantic equivalence.

We list the fundamental equivalences which form the basis of an evaluation strategy, and they are also useful for equational reasoning about effectful computations. First, we have the equivalences governing handlers that were informally explained in §2.2. If $h$ is the handler $\operatorname{handler} \operatorname{val} x \mapsto \rightarrow c_v | (\iota \# \operatorname{op} \iota x_i \mapsto \rightarrow c_i)_i$ then

$$\text{with } h \text{ handle } (\text{val } e) \equiv c_v[x \mapsto e]$$

and

$$\text{with } h \text{ handle } (\text{let } y = \iota \# \operatorname{op} \iota e \text{ in } c) \equiv c_i[x_i \mapsto e, k_i \mapsto (\text{fun } y \mapsto \text{with } h \text{ handle } c)]$$

and, assuming $\iota \neq \iota_i$ for all $i$,

$$\text{with } h \text{ handle } (\text{let } y = \iota \# \operatorname{op} \iota e \text{ in } c) \equiv \text{let } y = \iota \# \operatorname{op} \iota e \text{ in } (\text{with } h \text{ handle } c).$$

These equations were identified before by [16]. A variety of other equivalences is readily validated, such as $\beta\eta$-conversions for functions and the “associativity” law [12] ($y$ must not occur freely in $c_3$)

$$\text{let } x = (\text{let } y = c_1 \text{ in } c_2) \text{ in } c_3 \equiv \text{let } y = c_1 \text{ in } (\text{let } x = c_2 \text{ in } c_3).$$
The proof of this law proceeds by induction, as it uses the inductive nature of the per associated with the type of \( c_1 \). The inductive nature of the argument is similar to the one given by [17].

4 Discussion

We have presented a fairly simple effect system, which nevertheless allows us to deduce non-trivial properties of computations. There are several aspects of the system which we would like and plan to improve.

First, the per model is clearly suggesting that parametric polymorphism should blend naturally with the effect system. Once this is done we may have to pass to a parametric rather than the naive per model.

Second, in full Eff instances may be created dynamically with \texttt{new E}. To account for these in semantics, we would have to further complicate both the effect system and the semantics so that they could account for freshness phenomena. This would presumably follow the work of [13][19].

Third, we wrote the typing rules so that they are suitable for effect checking, but we really want effect inference. Initial experiments and a prototype implementation suggest that this should be not only doable, but quite likely useful too.

References

[1] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. arXiv:1203.1539, 2012.

[2] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In APPSEM 2000, volume 2395 of Lecture Notes in Computer Science, pages 42–122, 2000.

[3] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In ESOP’88, pages 94–114. Springer, 1988.

[4] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. Theoretical Computer Science, 357(1–3):70–99, 2006.

[5] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. The 1st ACM SIGPLAN Workshop on Higher-Order Programming with Effects, 2012.

[6] Ohad Kammar and Gordon D. Plotkin. Algebraic foundations for effect-dependent optimisations. In John Field and Michael Hicks, editors, POPL, pages 349–360. ACM, 2012.

[7] Oleg Kiselyov. Lazy IO breaks equational reasoning. April 2013. http://okmij.org/ftp/Haskell/index.html\char35 lazyIO-not-True

[8] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system (release 3.12): Documentation and user’s manual. Institut National de Recherche en Informatique et en Automatique, 2011.
[9] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185:182–210, September 2003.

[10] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL’88, pages 47–57, New York, NY, USA, 1988. ACM.

[11] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[12] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[13] Gabbay J. Murdoch and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, July 2001.

[14] Gordon Plotkin and John Power. Notions of computation determine monads. In *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356, 2002.

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

[16] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94, 2009.

[17] Matija Pretnar. *The Logic and Handling of Algebraic Effects*. PhD thesis, School of Informatics, University of Edinburgh, 2010.

[18] John Reynolds. The meaning of types—from intrinsic to extrinsic semantics. Technical report, Department of Computer Science, University of Aarhus, 2000.

[19] Ian Stark. Categorical models for local names. *LISP and Symbolic Computation*, 9(1):77–107, February 1996.

[20] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of functional programming*, 2(3):245–271, 1992.

[21] Philip Wadler. The marriage of effects and monads. *ACM SIGPLAN Notices*, 34(1):63–74, 1999.