Programming with Algebraic Effects and Handlers

Andrej Bauer
andrej@andrej.com

Matija Pretnar
matija@pretnar.info

Department of Mathematics and Physics
University of Ljubljana, Slovenia

Abstract

Eff is a programming language based on the algebraic approach to computational effects, in which effects are viewed as algebraic operations and effect handlers as homomorphisms from free algebras. *Eff* supports first-class effects and handlers through which we may easily define new computational effects, seamlessly combine existing ones, and handle them in novel ways. We give a denotational semantics of *eff* and discuss a prototype implementation based on it. Through examples we demonstrate how the standard effects are treated in *eff*, and how *eff* supports programming techniques that use various forms of delimited continuations, such as backtracking, breadth-first search, selection functionals, cooperative multi-threading, and others.

Introduction

Eff is a programming language based on the algebraic approach to effects, in which computational effects are modelled as operations of a suitably chosen algebraic theory [12]. Common computational effects such as input, output, state, exceptions, and non-determinism, are of this kind. Continuations are not algebraic [4], but they turn out to be naturally supported by *eff* as well. Effect handlers are a related notion [14, 17] which encompasses exception handlers, stream redirection, transactions, backtracking, and many others. These are modelled as homomorphisms induced by the universal property of free algebras.

Because an algebraic theory gives rise to a monad [11], algebraic effects are subsumed by the monadic approach to computational effects [1]. They have their own virtues, though. Effects are combined more easily than monads [5], and the interaction between effects and handlers offers new ways of programming. An experiment in the design of a programming language based on the algebraic approach therefore seems warranted.

Philip Wadler once opined [19] that monads as a programming concept would not have been discovered without their category-theoretic counterparts, but once they were, programmers could live in blissful ignorance of their origin. Because the same holds

for algebraic effects and handlers, we streamlined the paper for the benefit of programmers, trusting that connoisseurs will recognize the connections with the underlying mathematical theory.

The paper is organized as follows. Section 1 describes the syntax of *eff*, Section 2 informally introduces constructs specific to *eff*, Section 3 is devoted to type checking, in Section 4 we give a domain-theoretic semantics of *eff*, and in Section 5 we briefly discuss our prototype implementation. The examples in Section 6 demonstrate how effects and handlers can be used to produce standard computational effects, such as exceptions, state, input and output, as well as their variations and combinations. Further examples show how *eff*'s delimited control capabilities are used for nondeterministic and probabilistic choice, backtracking, selection functionals, and cooperative multithreading. We conclude with thoughts about the future work.

The implementation of *eff* is freely available at `http://math.andrej.com/eff/`.

Acknowledgements We thank Ohad Kammar, Gordon Plotkin, Alex Simpson, and Chris Stone for helpful discussions and suggestions. Ohad Kammar contributed parts of the type inference code in our implementation of *eff*. The cooperative multithreading example from Section 6.10 was written together with Chris Stone.

1 Syntax

Eff is a statically typed language with parametric polymorphism and type inference. Its types include products, sums, records, and recursive type definitions. To keep to the point, we focus on a core language with monomorphic types and type checking. The concrete syntax follows that of OCaml [6], and except for new constructs, we discuss it only briefly.

1.1 Types

Apart from the standard types, *eff* has *effect types* E and *handler types* $A \Rightarrow B$:

$$\begin{aligned} A, B, C &::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{empty} \mid && \text{(type)} \\ &\qquad A \times B \mid A + B \mid A \to B \mid E \mid A \Rightarrow B, \\ E &::= \texttt{effect}\ (\texttt{operation}\ \texttt{op}_i : A_i \to B_i)_i\ \texttt{end}. && \text{(effect type)} \end{aligned}$$

In the rule for effect types and elsewhere below $(\cdots)_i$ indicates that $\cdots$ may be repeated a finite number of times. We include the empty type as we need it to describe exceptions, see Section 6.2. An effect type describes a collection of related operation symbols, for example those for writing to and reading from a communication channel. We write $\texttt{op} : A \to B \in E$ or just $\texttt{op} \in E$ to indicate that the effect type E contains an operation `op` with parameters of type A and results of type B. The handler type $A \Rightarrow B$ should be understood as the type of handlers acting on computations of type A and yielding computations of type B.

1.2 Expressions and computations

Eff distinguishes between *expressions* and *computations*, which are similar to values and producers of fine-grain call-by-value [7]. The former are inert and free from computational effects, including divergence, while the latter may diverge or cause computational effects. As discussed in Section 5, the concrete syntax of *eff* hides the distinction and allows the programmer to freely mix expressions and computations.

Beware that we use two kinds of vertical bars below: the tall $\mid$ separates grammatical alternatives, and the short | separates cases in handlers and match statements. The expressions are

$$\begin{aligned} e \;::=\;& x \mid n \mid c \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{(}e_1\texttt{,}\, e_2\texttt{)} \mid && \text{(expression)} \\ & \texttt{Left}\; e \mid \texttt{Right}\; e \mid \texttt{fun}\; x : A \mapsto c \mid e\,\texttt{\#}\,\texttt{op} \mid h, \\ h \;::=\;& \texttt{handler}\; (e_i\,\texttt{\#}\,\texttt{op}_i\, x\, k \mapsto c_i)_i \mid \texttt{val}\; x \mapsto c_v \mid \texttt{finally}\; x \mapsto c_f, && \text{(handler)} \end{aligned}$$

where x signifies a variable, n an integer constant, and c other built-in constants. The expressions `()`, $\texttt{(}e_1\texttt{,}\, e_2\texttt{)}$, `Left` e, `Right` e, and `fun` $x : A \mapsto c$ are introduction forms for the unit, product, sum, and function types, respectively. Operations $e\,\texttt{\#}\,\texttt{op}$ and handlers h are discussed in Section 2.

The computations are

$$\begin{aligned} c \;::=\;& \texttt{val}\; e \mid \texttt{let}\; x = c_1\; \texttt{in}\; c_2 \mid \texttt{let rec}\; f\, x = c_1\; \texttt{in}\; c_2 \mid && \text{(computation)} \\ & \texttt{if}\; e\; \texttt{then}\; c_1\; \texttt{else}\; c_2 \mid \texttt{match}\; e\; \texttt{with} \mid \texttt{match}\; e\; \texttt{with}\; \texttt{(}x\texttt{,}\, y\texttt{)} \mapsto c \mid \\ & \texttt{match}\; e\; \texttt{with}\; \texttt{Left}\; x \mapsto c_1 \mid \texttt{Right}\; y \mapsto c_2 \mid e_1\, e_2 \mid \\ & \texttt{new}\; E \mid \texttt{new}\; E\; @\; e\; \texttt{with}\; (\texttt{operation}\; \texttt{op}_i\, x\, @\, y \mapsto c_i)_i\; \texttt{end} \mid \\ & \texttt{with}\; e\; \texttt{handle}\; c. \end{aligned}$$

An expression e is promoted to a computation with `val` e, but in the concrete syntax `val` is omitted, as there is no distinction between expressions and computations. The statement `let` $x = c_1$ `in` c_2 binds x in c_2, and `let rec` $f\, x = c_1$ `in` c_2 defines a recursive function f in c_2. The conditional statement and the variations of `match` are elimination forms for booleans, the empty type, products, and sums, respectively. Instance creation and the handling construct are discussed in Section 2.

Arithmetical expressions such as $e_1 + e_2$ count as computations because the arithmetical operators are defined as built-in constants, so that $e_1 + e_2$ is parsed as a double application. This allows us to uniformly treat all operations, irrespective of whether they are pure or effectful (division by zero).

2 Constructs specific to *eff*

We explain the intuitive meaning of notions that are specific to *eff*, namely instances, operations, handlers, and resources. We allow ourselves some slack in distinguishing syntax from semantics, which is treated in detail in Section 4. It is helpful to think of a terminating computation as evaluating either to a value or an operation applied to a parameter.

2.1 Instances and operations

The computation `new` E generates a fresh *effect instance* of effect type E. For example, `new ref` generates a new reference, `new channel` a new communication channel, etc. The extended form of `new` creates an effect instance with an associated *resource*, which determines the default behaviour of operations and is explained separately in Section 2.3.

For each effect instance e of effect type E and an operation symbol $\mathtt{op} \in E$ there is an *operation* $e \,\#\, \mathtt{op}$, also known as a *generic effect* [12]. By itself, an operation is a value, and hence effect-free, but an applied operation $e \,\#\, \mathtt{op}\, e'$ is a computational effect whose ramifications are determined by enveloping handlers and the resource associated with e.

2.2 Handlers

A handler

$$h = \mathtt{handler}\ (e_i \,\#\, \mathtt{op}_i\, x\, k \mapsto c_i)_i \mid \mathtt{val}\ x \mapsto c_v \mid \mathtt{finally}\ x \mapsto c_f$$

may be applied to a computation c with the handling construct

$$\mathtt{with}\ h\ \mathtt{handle}\ c, \tag{1}$$

which works as follows (we ignore the `finally` clause for the moment):

1. If c evaluates to `val` e, (1) evaluates to c_v with x bound to e.
2. If the evaluation of c encounters an operation $e_i \,\#\, \mathtt{op}_i\, e$, (1) evaluates to c_i with x bound to e and k bound to the continuation of $e_i \,\#\, \mathtt{op}_i\, e$, i.e., whatever remains to be computed after the operation. The continuation is delimited by (1) and is handled by h as well.

The `finally` clause can be thought of as an outer wrapper which performs an additional transformation, so that (1) is equivalent to

$$\mathtt{let}\ x = (\mathtt{with}\ h'\ \mathtt{handle}\ c)\ \mathtt{in}\ c_f$$

where h' is like h without the `finally` clause. Such a wrapper is useful because we often perform the same transformation every time a given handler is applied. For example, the handler for state handles a computation by transforming it to a function accepting the state, and `finally` applies the function to the initial state, see Section 6.3.

If the evaluation of c encounters an operation $e \,\#\, \mathtt{op}\, e'$ that is not listed in h, the control propagates to outer handling constructs, and eventually to the toplevel, where the behaviour is determined by the resource associated with e.

2.3 Resources

The construct

$$\mathtt{new}\ E \mathbin{@} e\ \mathtt{with}\ (\mathtt{operation}\ \mathtt{op}_i\, x \mathbin{@} y \mapsto c_i)_i\ \mathtt{end}$$

creates an instance n with an associated *resource*, inspired by coalgebraic semantics of computational effects [16, 13]. A resource carries a state and prescribes the default behaviour of the operations $n\,\#\,\mathtt{op}_i$. The paradigmatic case of resources is the definition of ML-style references, see Section 6.3.

The initial resource state for n is set to e. When the toplevel evaluation encounters an operation $n\,\#\,\mathtt{op}_i\,e'$, it evaluates c_i with x bound to e' and y bound to the current resource state. The result must be a pair of values, the first of which is passed to the continuation, and the second of which is the new resource state. If c_i evaluates to an operation rather than a pair of values, a runtime error is reported, as there is no reasonable way of handling it.

In *eff* the interaction with the real world is accomplished through built-in resources. For example, there is a predefined channel instance `std` with operations `std#read` and `std#write` whose associated resource performs actual interaction with the standard input and the standard output.

3 Type checking

Types in *eff* are like those of ML [9] in the sense that they do not capture any information about computational effects. There are two typing judgements, $\Gamma \vdash_{\mathsf{e}} e : A$ states that expression e has type A in context Γ, and $\Gamma \vdash_{\mathsf{c}} c : A$ does so for a computation c. As usual, a context Γ is a list of distinct variables with associated types. The standard typing rules for expressions are:

$$\frac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{e}} x : A} \qquad \Gamma \vdash_{\mathsf{e}} n : \mathtt{int} \qquad \Gamma \vdash_{\mathsf{e}} \mathtt{true} : \mathtt{bool} \qquad \Gamma \vdash_{\mathsf{e}} \mathtt{false} : \mathtt{bool}$$

$$\Gamma \vdash_{\mathsf{e}} () : \mathtt{unit} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e_1 : A \qquad \Gamma \vdash_{\mathsf{e}} e_2 : B}{\Gamma \vdash_{\mathsf{e}} (e_1, e_2) : A \times B} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e : A}{\Gamma \vdash_{\mathsf{e}} \mathtt{Left}\ e : A + B}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : B}{\Gamma \vdash_{\mathsf{e}} \mathtt{Right}\ e : A + B} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{c}} c : B}{\Gamma \vdash_{\mathsf{e}} \mathtt{fun}\ x : A \mapsto c : A \to B}$$

We also have to include judgements that assign types to other built-in constants. An operation receives a function type

$$\frac{\Gamma \vdash_{\mathsf{e}} e : E \qquad \mathtt{op} : A \to B \in E}{\Gamma \vdash_{\mathsf{e}} e\,\#\,\mathtt{op} : A \to B}$$

while a handler is typed by the somewhat complicated rule

$$\frac{\Gamma \vdash_{\mathsf{e}} e_i : E_i \qquad \mathtt{op}_i : A_i \to B_i \in E_i \qquad \Gamma, x : A_i, k : B_i \to B \vdash_{\mathsf{c}} c_i : B \qquad \Gamma, x : A \vdash_{\mathsf{c}} c_v : B \qquad \Gamma, x : B \vdash_{\mathsf{c}} c_f : C}{\Gamma \vdash_{\mathsf{e}} (\mathtt{handler}\ (e_i\,\#\,\mathtt{op}_i\,x\,k \mapsto c_i)_i \mid \mathtt{val}\ x \mapsto c_v \mid \mathtt{finally}\ x \mapsto c_f) : A \Rightarrow C}$$

which states that a handler first handles a computation of type A into a computation of type B according to the `val` and operation clauses, after which the `finally` clause transforms it further into a computation of type C.

The typing rules for computations are familiar or expected. Promotions of expressions and `let` statements are typed by

$$\frac{\Gamma \vdash_{\mathsf{e}} e : A}{\Gamma \vdash_{\mathsf{c}} \mathtt{val}\ e : A} \qquad \frac{\Gamma \vdash_{\mathsf{c}} c_1 : A \qquad \Gamma, x : A \vdash_{\mathsf{c}} c_2 : B}{\Gamma \vdash_{\mathsf{c}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : B}$$

$$\frac{\Gamma, f : A \to B, x : A \vdash_{\mathsf{c}} c_1 : B \qquad \Gamma, f : A \to B \vdash_{\mathsf{c}} c_2 : C}{\Gamma \vdash_{\mathsf{c}} \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 : C}$$

and various elimination forms are typed by

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \mathtt{bool} \qquad \Gamma \vdash_{\mathsf{c}} c_1 : A \qquad \Gamma \vdash_{\mathsf{c}} c_2 : A}{\Gamma \vdash_{\mathsf{c}} \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : A} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e : \mathtt{empty}}{\Gamma \vdash_{\mathsf{c}} \mathtt{match}\ e\ \mathtt{with} : A}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : A \times B \qquad \Gamma, x : A, y : B \vdash_{\mathsf{c}} c : C}{\Gamma \vdash_{\mathsf{c}} \mathtt{match}\ e\ \mathtt{with}\ (x, y) \mapsto c : C}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : A + B \qquad \Gamma, x : A \vdash_{\mathsf{c}} c_1 : C \qquad \Gamma, y : B \vdash_{\mathsf{c}} c_2 : C}{\Gamma \vdash_{\mathsf{c}} \mathtt{match}\ e\ \mathtt{with}\ \mathtt{Left}\ x \mapsto c_1 \mid \mathtt{Right}\ y \mapsto c_2 : C}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e_1 : A \to B \qquad \Gamma \vdash_{\mathsf{e}} e_2 : A}{\Gamma \vdash_{\mathsf{c}} e_1\, e_2 : B}$$

The instance creation is typed by the rules

$$\Gamma \vdash_{\mathsf{c}} \mathtt{new}\ E : E$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : C \qquad \mathtt{op}_i : A_i \to B_i \in E \qquad \Gamma, x : A_i, y : C \vdash_{\mathsf{c}} c_i : B_i \times C}{\Gamma \vdash_{\mathsf{c}} \mathtt{new}\ E\ @\ e\ \mathtt{with}\ (\mathtt{operation}\ \mathtt{op}_i\ x\ @\ y \mapsto c_i)_i\ \mathtt{end} : E}$$

The rule for the simple form is obvious, while the one for the extended form checks that the initial state e has type C and that, for each operation $\mathtt{op}_i : A_i \to B_i \in E$, the corresponding computation c_i evaluates to a pair of type $B_i \times C$.

Finally, the rule for handling expresses the fact that handlers are like functions:

$$\frac{\Gamma \vdash_{\mathsf{c}} c : A \qquad \Gamma \vdash_{\mathsf{e}} e : A \Rightarrow B}{\Gamma \vdash_{\mathsf{c}} \mathtt{with}\ e\ \mathtt{handle}\ c : B}$$

4 Denotational semantics

Our aim is to describe a denotational semantics which explains how programs in *eff* are evaluated. Since the implemented runtime has no type information, we give Curry-style semantics in which terms are interpreted without being typed. See the exposition by John Reynolds [18] on how such semantics can be related to Church-style semantics in which types and typing judgements receive meanings.

We give interpretations of expressions and computations in domains of *values* V and *results* R, respectively. We follow Reynolds by avoiding a particular choice of V and R, and instead require properties of V and R that ensure the semantics works out. The requirements can be met in a number of ways, for example by solving suitable domain equations or by taking V and R to be sufficiently large universal domains.

The domain V has to contain integers, booleans, functions, etc. In particular, we require that V contains the following retracts, where $\mathbb{I}$ is a set of effect instances, and $\oplus$ is coalesced sum:

$$\mathbb{Z}_\bot \underset{\rho_{\mathtt{int}}}{\overset{\iota_{\mathtt{int}}}{\rightleftarrows}} V \qquad \{0,1\}_\bot \underset{\rho_{\mathtt{bool}}}{\overset{\iota_{\mathtt{bool}}}{\rightleftarrows}} V \qquad \{\star\}_\bot \underset{\rho_{\mathtt{unit}}}{\overset{\iota_{\mathtt{unit}}}{\rightleftarrows}} V$$

$$\mathbb{I}_\bot \underset{\rho_{\mathtt{effect}}}{\overset{\iota_{\mathtt{effect}}}{\rightleftarrows}} V \qquad V \times V \underset{\rho_\times}{\overset{\iota_\times}{\rightleftarrows}} V \qquad V \oplus V \underset{\rho_+}{\overset{\iota_+}{\rightleftarrows}} V$$

$$R^V \underset{\rho_\to}{\overset{\iota_\to}{\rightleftarrows}} V \qquad R^R \underset{\rho_\Rightarrow}{\overset{\iota_\Rightarrow}{\rightleftarrows}} V$$

As expressions are terminating, the bottom element of V is never used to denote divergence, but we do use it to indicate ill-formed values and runtime errors.

The domain

$$(V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \tag{2}$$

embodies the idea that a terminating computation is either a value or an operation applied to a parameter and a continuation. There are canonical retractions from (2) onto the two summands,

$$V \underset{\rho_{\mathtt{val}}}{\overset{\iota_{\mathtt{val}}}{\rightleftarrows}} (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \underset{\rho_{\mathtt{oper}}}{\overset{\iota_{\mathtt{oper}}}{\leftrightarrows}} (\mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \tag{3}$$

A typical element of (2) is either $\bot$, of the form $\iota_{\mathtt{val}}(v)$ for a unique $v \in V$, or of the form $\iota_{\mathtt{oper}}(n, \mathtt{op}, v, \kappa)$ for unique $n \in \mathbb{I}$, $\mathtt{op} \in \mathbb{O}$, $v \in V$, and $\kappa \in R^V$. We require that R contains (2) as a retract:

$$(V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \underset{\rho_{\mathtt{res}}}{\overset{\iota_{\mathtt{res}}}{\leftrightarrows}} R. \tag{4}$$

We may define a strict map from (2) by cases, with one case specifying how to map $\iota_{\mathtt{val}}(v)$ and the other how to map $\iota_{\mathtt{oper}}(n, \mathtt{op}, v, \kappa)$. For example, given a map $f : V \to R$, there is a unique strict map $f^\dagger : (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \to R$, called the *lifting* of f, which depends on f continuously and satisfies the recursive equations

$$f^\dagger(\iota_{\mathtt{val}}(v)) = f(v),$$
$$f^\dagger(\iota_{\mathtt{oper}}(n, \mathtt{op}, v, \kappa)) = \iota_{\mathtt{oper}}(n, \mathtt{op}, v, f^\dagger \circ \rho_{\mathtt{res}} \circ \kappa).$$

An *environment* η 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 η. An expression is

interpreted as a map from environments to values. The standard cases are as follows:

$$\begin{aligned}
[\![x]\!]\,\eta &= \eta(x) \\
[\![n]\!]\,\eta &= \iota_{\mathtt{int}}(\overline{n}) \\
[\![\mathtt{false}]\!]\,\eta &= \iota_{\mathtt{bool}}(0) \\
[\![\mathtt{true}]\!]\,\eta &= \iota_{\mathtt{bool}}(1) \\
[\![\mathtt{()}]\!]\,\eta &= \iota_{\mathtt{unit}}(\star) \\
[\![\mathtt{(}e_1, e_2\mathtt{)}]\!]\,\eta &= \iota_{\times}([\![e_1]\!]\,\eta, [\![e_2]\!]\,\eta) \\
[\![\mathtt{Left}\ e]\!]\,\eta &= \iota_{+}(\iota_0([\![e]\!]\,\eta)) \\
[\![\mathtt{Right}\ e]\!]\,\eta &= \iota_{+}(\iota_1([\![e]\!]\,\eta)) \\
[\![\mathtt{fun}\ x : A \mapsto c]\!]\,\eta &= \iota_{\to}(\lambda v : V\,.\,[\![c]\!]\,\eta[x \mapsto v])
\end{aligned}$$

Of course, we need to provide the semantics of other built-in constants, too. The interpretation of $e\,\mathtt{\#\,op}$ make sense only when e evaluates to an instance, so we define

$$[\![e\,\mathtt{\#\,op}]\!]\,\eta = \begin{cases} \iota_{\to}(\lambda v : V\,.\,\iota_{\mathtt{res}}(\iota_{\mathtt{oper}}(n, \mathtt{op}, v, \iota_{\mathtt{res}} \circ \iota_{\mathtt{val}}))) & \text{if } \rho_{\mathtt{effect}}([\![e]\!]\,\eta) = n \in \mathbb{I}, \\ \iota_{\to}(\lambda v : V\,.\,\bot) & \text{if } \rho_{\mathtt{effect}}([\![e]\!]\,\eta) = \bot. \end{cases}$$

The interpretation of a handler is

$$[\![\mathtt{handler}\ (e_i\,\mathtt{\#\,op}_i\,x\,k \mapsto c_i)_i\ \mathtt{|}\ \mathtt{val}\ x \mapsto c_v\ \mathtt{|}\ \mathtt{finally}\ x \mapsto c_f]\!]\,\eta = \iota_{\Rightarrow}(f^{\dagger} \circ \rho_{\mathtt{res}} \circ h \circ \rho_{\mathtt{res}})$$

where $f : V \to R$ is $f(v) = [\![c_f]\!]\,\eta[x \mapsto v]$ and $h : (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_{\bot} \to R$ is characterized as follows: if one of the $\rho_{\mathtt{effect}}([\![e_i]\!]\,\eta)$ is $\bot$ we set $h = \lambda x\,.\,\bot$, otherwise $\rho_{\mathtt{effect}}([\![e_i]\!]\,\eta) = n_i \in \mathbb{I}$ for all i and then we take the h defined by cases as

$$\begin{aligned}
h(\iota_{\mathtt{val}}(v)) &= [\![c_v]\!]\,\eta[x \mapsto v] \\
h(\iota_{\mathtt{oper}}(n_i, \mathtt{op}_i, v, \kappa)) &= [\![c_i]\!]\,\eta[x \mapsto v, k \mapsto \kappa] \qquad \text{for all } i, \\
h(\iota_{\mathtt{oper}}(n, \mathtt{op}, v, \kappa)) &= \iota_{\mathtt{res}}(\iota_{\mathtt{oper}}(n, \mathtt{op}, v, h \circ \rho_{\mathtt{res}} \circ \kappa)) \\
&\qquad \text{if } (n, \mathtt{op}) \neq (n_i, \mathtt{op}_i) \text{ for all } i.
\end{aligned}$$

We proceed to the meaning of computations, which are interpreted as maps from environments to results. Promotion of expressions is interpreted in the obvious way as

$$[\![\mathtt{val}\ e]\!]\,\eta = \iota_{\mathtt{res}}(\iota_{\mathtt{val}}([\![e]\!]\,\eta))$$

The `let` statement corresponds to monadic-style binding:

$$[\![\mathtt{let}\ x = c_1\ \mathtt{in}\ c_2]\!]\,\eta = (\lambda v : V\,.\,[\![c_2]\!]\,\eta[x \mapsto v])^{\dagger}(\rho_{\mathtt{res}}([\![c_1]\!]\,\eta)),$$

A recursive function definition is interpreted as

$$[\![\mathtt{let\ rec}\ f\,x = c_1\ \mathtt{in}\ c_2]\!]\,\eta = [\![c_2]\!]\,\eta[f \mapsto \iota_{\to}(t)]$$

where $t : V \to R$ is the least fixed point of the map

$$t \mapsto (\lambda v : V \,.\, [\![c_1]\!]\,\eta[f \mapsto \iota_{\to}(t), x \mapsto v]).$$

The elimination forms are interpreted in the usual way as:

$$[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\,\eta = \begin{cases} [\![c_1]\!]\,\eta & \text{if } \rho_{\texttt{bool}}[\![e]\!]\,\eta = 1 \\ [\![c_2]\!]\,\eta & \text{if } \rho_{\texttt{bool}}[\![e]\!]\,\eta = 0 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\texttt{match}\ e\ \texttt{with}]\!]\,\eta = \bot$$

$$[\![\texttt{match}\ e\ \texttt{with}\ \texttt{(}x, y\texttt{)} \mapsto c]\!]\,\eta = [\![c]\!]\,\eta[x \mapsto v_0, y \mapsto v_1] \quad \text{where } (v_0, v_1) = \rho_{\times}([\![e]\!]\,\eta)$$

$$[\![\texttt{match}\ e\ \texttt{with}\ \texttt{Left}\ x \mapsto c_1\ \texttt{|}\ \texttt{Right}\ y \mapsto c_2]\!]\,\eta = \begin{cases} [\![c_1]\!]\,\eta[x \mapsto v] & \text{if } \rho_{+}([\![e]\!]\,\eta) = \iota_0(v) \\ [\![c_2]\!]\,\eta[y \mapsto v] & \text{if } \rho_{+}([\![e]\!]\,\eta) = \iota_1(v) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![e_1\, e_2]\!]\,\eta = \rho_{\to}([\![e_1]\!]\,\eta)([\![e_2]\!]\,\eta)$$

For the interpretation of `new` we need a way of generating fresh names so that we may sensibly interpret

$$[\![\texttt{new}\ E]\!]\,\eta = \iota_{\texttt{res}}(\iota_{\texttt{val}}(\iota_{\texttt{effect}}(n))) \quad \text{where } n \in \mathbb{I} \text{ is fresh.}$$

The implementation simply uses a local counter, but a satisfactory semantic solution needs a model of names, such as the permutation models of Pitts and Gabbay [10], with $\mathbb{I}$ then being the set of atoms.

Finally, the handling construct is just an application

$$[\![\texttt{with}\ e\ \texttt{handle}\ c]\!]\,\eta = \rho_{\Rightarrow}([\![e]\!]\,\eta)([\![c]\!]\,\eta).$$

4.1 Semantics of resources

To model resources, the denotational semantics has to support the mutable nature of resource state, for example by explicitly threading it through the evaluation. But we prefer not to burden ourselves and the readers with the ensuing technicalities. Instead, we assume a mutable store σ indexed by effect instances which supports the lookup and update operations. That is, $\sigma[n]$ gives the current contents at location $n \in \mathbb{I}$, while $\sigma[n] \leftarrow\!\!\shortmid v\,;\, x$ sets the contents at location n to $v \in V$ and yields x.

A resource describes the behaviour of operations, i.e., it is a map $\mathbb{O} \times V \times V \to V \times V$ which computes a value and the new state from a given operation symbol, parameter, and the current state. Thus an effect instance is not just an element of $\mathbb{I}$ anymore, but an element of $\mathbb{J} = \mathbb{I} \times (V \times V)^{\mathbb{O} \times V \times V}$. Consequently in the semantics we replace $\mathbb{I}$ with $\mathbb{J}$ throughout and adapt the semantics of `new` so that it sets the initial resource state and gives an element of $\mathbb{J}$:

$$[\![\texttt{new}\ E\ @\ e\ \texttt{with}\ (\texttt{operation}\ \texttt{op}_i\ x\ @\ y \mapsto c_i)_i\ \texttt{end}]\!]\,\eta = \sigma[n] \leftarrow\!\!\shortmid [\![e]\!]\,\eta\,;\, \iota_{\texttt{res}}(\iota_{\texttt{val}}(\iota_{\texttt{effect}}(n, r))) \quad \text{where } n \in \mathbb{I} \text{ is fresh}$$

where $r : \mathbb{O} \times V \times V \to V \times V$ is defined by

$$r(\mathtt{op}, v, s) = \begin{cases} \rho_\times(\rho_{\mathtt{val}}(\rho_{\mathtt{res}}([\![c_i]\!]\,\eta[x \mapsto v, y \mapsto s]))) & \text{if } \mathtt{op} = \mathtt{op}_i \text{ for some } i, \\ \bot & \text{otherwise.} \end{cases}$$

If no resource is provided, we use a trivial one:

$$[\![\mathtt{new}\ E]\!]\,\eta = \iota_{\mathtt{res}}(\iota_{\mathtt{val}}(\iota_{\mathtt{effect}}(n, \bot))) \qquad \text{where } n \in \mathbb{I} \text{ is fresh.}$$

Finally, to model evaluation at the toplevel, we define $\mathcal{E} : (V + \mathbb{I} \times \mathbb{O} \times V \times R^V)_\bot \to V$ by cases:

$$\begin{aligned} \mathcal{E}(\iota_{\mathtt{val}}\,(v)) &= v \\ \mathcal{E}(\iota_{\mathtt{oper}}((n, r), \mathtt{op}, v, \kappa)) &= \sigma[n] \leftarrow s\,;\, \mathcal{E}(\rho_{\mathtt{res}}(\kappa\, v')) \\ &\qquad \text{where } (v', s) = r(\mathtt{op}, v, \sigma[n]). \end{aligned}$$

The meaning of a computation c at the toplevel in the environment η (and an implicit resource state σ) is $\mathcal{E}(\rho_{\mathtt{res}}([\![c]\!]\,\eta))$.

5 Implementation

To experiment with *eff* we implemented a prototype interpreter whose main evaluation loop is essentially the same as the denotational semantics described in Section 4. Apart from inessential improvements, such as recursive type definitions, inclusion of `for` and `while` loops, and pattern matching, the implemented language differs from the one presented here in two ways that make it usable: we implemented Hindley-Milner style type inference with parametric polymorphism [8], and in the concrete syntax we hid the distinction between expressions and computations. We briefly discuss each.

There are no surprises in the passage from monomorphic type checking to parametric polymorphism and type inference. The infamous value restriction [20] is straightforward because the distinction between expressions and computations corresponds exactly to the distinction between nonexpansive and expansive terms. In fact, it may be worth investing in an effect system that would relax the value restriction on those computations that can safely be presumed pure. Because `new` E is a computation, effect instances are *not* polymorphic, which is in agreement with ML-style references being non-polymorphic.

A syntactic division between pure expressions and possibly effectful computations is annoying because even something as simple as $f\,x\,y$ has to be written as `let` $g = f\,x$ `in` $g\,y$, and having to insert `val` in the right places is no fun either. Therefore, the concrete syntax allows the programmer to arbitrarily mix expressions and computations, and a desugaring phase appropriately separates the two.

The desugaring process is fairly simple. It inserts a `val` when an expression stands where a computation is expected. And if a computation stands where an expression is expected, the computation is hoisted to an enclosing `let` statement. Because several

computations may be hoisted from a single expression, the question arises how to order the corresponding `let` statements. For example, $(f\,x, g\,y)$ can be desugared as

$$\texttt{let}\ a = f\,x\ \texttt{in}\ \texttt{let}\ b = g\,y\ \texttt{in}\ (a, b) \quad \text{or} \quad \texttt{let}\ b = g\,y\ \texttt{in}\ \texttt{let}\ a = f\,x\ \texttt{in}\ (a, b)$$

The order of $f\,x$ and $g\,y$ matters when both computations cause computational effects. The desugaring phase avoids making a decision by using the *simultaneous* `let` statement

$$\texttt{let}\ a = f\,x\ \texttt{and}\ b = g\,y\ \texttt{in}\ (a, b)$$

which leaves open the possibility of various compiler optimizations. The prototype simply evaluates simultaneous bindings in the order they are given, and a command-line option enables sequencing warnings about possible unexpected order of effects. It could be argued that the warnings should actually be errors, but we allow some slack until we have an effect system that can detects harmless instances of simultaneous binding.

For one-off handlers, *eff* provides an inline syntax so that one can write

```
handle
  c
with
  ...
```

instead of

```
with
  handler
    ...
handle
  c
```

Additionally, the `val` and `finally` clauses may be omitted, in which case they are assumed to be the identities.

6 Examples

In this section we consider a number of examples that demonstrate the possibilities offered by first-class effects and handlers. Functional programmers will notice similarities with the monadic programming style, and continuations aficionados will recognize their favourite tricks as well. The point we are making though is that even though monads and continuations can be simulated in *eff*, it is usually more natural to use effects and handlers directly.

6.1 Choice

We start with an example that is infrequently met in practice but is a favourite of theoreticians, namely *(nondeterministic) choice*. A binary choice operation which picks a boolean value is described by an effect type with a single operation `decide`:

```
type choice = effect
  operation decide : unit -> bool
end
```

Let `c` be an effect instance of type `choice`:

```
let c = new choice
```

The computation

```
let x = (if c#decide () then 10 else 20) in
let y = (if c#decide () then 0 else 5) in
  x - y
```

expresses the fact that `x` receives either the value `10` or `20`, and `y` the value `0` or `5`. If we ran the above computation we would just get a message about an uncaught operation `c#decide`. For the computation to actually do something we need to wrap it in a handler. For example, if we want `c#decide` to always choose `true`, we handle the operation by passing `true` to the continuation `k`:

```
handle
  let x = (if c#decide () then 10 else 20) in
  let y = (if c#decide () then 0 else 5) in
    x - y
with
| c#decide () k -> k true
```

The result of course is `10`. A more interesting handler is one that collects all possible results. Because we are going to use it several times, we define a handler (the operator `@` is list concatenation):

```
let choose_all d = handler
  | d#decide () k -> k true @ k false
  | val x -> [x]
```

Notice that the handler calls the continuation `k` twice, once for each choice, and it concatenates the two lists so obtained. It also transforms a value to a singleton list. When we run

```
with choose_all c handle
  let x = (if c#decide () then 10 else 20) in
  let y = (if c#decide () then 0 else 5) in
    x - y
```

the result is the list `[10;5;20;15]`. Let us see what happens if we use two instances of `choice` with two handlers:

```
let c1 = new choice in
let c2 = new choice in
  with choose_all c1 handle
  with choose_all c2 handle
    let x = (if c1#decide () then 10 else 20) in
    let y = (if c2#decide () then 0 else 5) in
      x - y
```

Now the answer is `[[10;5];[20;15]]` because the outer handler runs the inner one twice, and the inner one produces a list of two possible results each time. If we switch the order of handlers *and* of operations,

```
let c1 = new choice in
let c2 = new choice in
  with choose_all c2 handle
  with choose_all c1 handle
    let y = (if c2#decide () then 0 else 5) in
    let x = (if c1#decide () then 10 else 20) in
      x - y
```

the answer is `[[10;20];[5;15]]`. For a true understanding of what is going on, the reader should figure out why we get a list of *four* lists, each containing two numbers, if we switch only the order of handlers but not of operations.

6.2 Exceptions

An exception is an effect with a single operation `raise` with an empty result type:

```
type 'a exception = effect
  operation raise : 'a -> empty
end
```

The parameter of `raise` carries additional data that can be used by an exception handler. The empty result type indicates that an exception, once raised, never yields the control back to the continuation. Indeed, as there are no expressions of the empty type (but there are of course computations of the empty type), a handler cannot restart the continuation of `raise`, which matches the standard behaviour of exception handlers.

In practice, most exception handlers are one-off and are written using the inline syntax discussed in Section 5. There are also convenient general exceptions handlers, for example,

```
let optionalize e = handler
  | e#raise _ _ -> None
  | val x -> Some x
```

converts a computation that possibly raises the given exception `e` to one that yields an optional result. We can use it as follows:

```
with optionalize e handle
  computation
```

In ML-style languages exceptions can be raised anywhere because `raise` is polymorphic, whereas in *eff* we cannot use e `#raise` e' freely because its type is empty, not polymorphic. This is rectified with the convenience function

```
let raise e x = match (e#raise x) with
```

of type α `exception` $\to \alpha \to \beta$ which eliminates the empty type, so we may use `raise` e e' anywhere.

Another difference between ML-style exceptions and those in *eff* is that the former are like a single instance of the latter, i.e., if we were to mimic ML exceptions in *eff* we would need a (dynamically extensible) datatype of exceptions `exc` and a single instance `e` of type `exc exception`. The definition of `raise` would be

```
let raise x = match (e#raise x) with
```

and exception handling would be done as usual. One consequence of this is that in ML it is possible to catch *all* exceptions at once, whereas in *eff* locally created exception instances are unreachable, just as local references are in ML. Which brings us to the next example.

6.3 State

In *eff* state is represented by a computational effect with operations for looking up and updating a value:

```
type 'a ref = effect
  operation lookup: unit -> 'a
  operation update: 'a -> unit
end
```

We refer to instances of type `ref` as *references*. To get the same behaviour as in ML, we handle them with

```
let state r x = handler
  | val y -> (fun s -> y)
  | r#lookup () k -> (fun s -> k s s)
  | r#update s' k -> (fun s -> k () s')
  | finally f -> f x
```

The handler passes the state around by converting computations to functions that accept the state. For example, lookup takes the state `s` and passes it to the continuation `k`. Because `k s` is handled too, it is again a function accepting state, so we pass `s` to `k s` again, which explains why we wrote `k s s`. Values and updates are handled in a similar fashion. The `finally` clause applies the function so obtained to the initial state `x`.

The above handler is impractical because for every use of a reference we have to repeat the idiom

```
let r = new ref in
  with state r x handle
     computation
```

An even bigger problem is that the reference may propagate outside the scope of its handler where its behaviour is undefined, for instance encapsulated in a λ-abstraction. The perfect solution to both problems is to use resources as follows:

```
let ref x =
  new ref @ x with
    operation lookup () @ s -> (s, s)
    operation update s' @ _ -> ((), s')
  end
```

With this definition a reference carries a current state which is initially set to `x`, lookup returns the current state without changing it, while update returns the unit and changes the state. With the definition of the operators

```
let (!) r = r#lookup ()
let (:=) r v = r#update v
```

we get *exactly* the ML syntax and behaviour. Of course, a particular reference may still be handled by a custom handler, for example to fetch its initial value from an external persistent storage and save the final value back into it.

6.4 Transactions

We may handle lookup and update so that the state remains unchanged in case an exception occurs. The handler which accomplishes this for a given reference `r` is

```
let transaction r = handler
  | r#lookup () k -> (fun s -> k s s)
  | r#update s' k -> (fun s -> k () s')
  | val x -> (fun s -> r := s; x)
  | finally f -> f !r
```

The handler passes around temporary state `s`, just like the `state` handler in Section 6.3, and only commits it to `r` when the handled computation terminates with a value. Thus the computation

```
with transaction r handle
  r := 23;
  raise e (3 * !r);
  r := 34
```

raises the exception `e` with parameter `69`, but does not change the value of `r`.

6.5 Deferred computations

There are many variations on store, of which we mention just one that can be implemented with resources, namely *lazy* or *deferred computations*. Such a computation is given by a thunk, i.e., a function whose domain is `unit`. If and when its value is needed, the thunk is *forced* by application to `()`, and the result is stored so that it can be given immediately upon subsequent forcing. This idea is embodied in the effect type

```
type 'a lazy = effect
  operation force: unit -> 'a
end
```

together with functions for creating and forcing deferred expressions:

```
type 'a deferred = Value of 'a | Thunk of (unit -> 'a)

let lazy t =
  new lazy @ (Thunk t) with
    operation force () @ v ->
      (match v with
        | Value v -> (v, Value v)
        | Thunk t -> let v = t () in (v, Value v))
```

```
  end

let force d = d#force ()
```

The function `lazy` takes a thunk `t` and creates a new instance whose initial state is `Thunk t`. The first time the instance is forced, the thunk is evaluated to a value `v`, and the state changes to `Value v`. Thereafter the stored value is returned immediately.

If the thunk triggers an operation, *eff* reports a runtime error because it does not allow operations in resources. While this may be seen as an obstacle, it also promotes good programming habits, for one should not defer effects to an unpredictable future time. It would be even better if deferred effectful computations were prevented by a typing discipline, but for that we would need an effect system.

6.6 Input and output

A program worth running has to connect with the real-world environment in some way. In *eff* this is done cleanly through built-in effect instances that provide an interface to the operating system. For input and output *eff* has a predefined effect type

```
type channel = effect
  operation read : unit -> string
  operation write : string -> unit
end
```

and a `channel` instance `std` which actually writes to standard output and reads from standard input. Of course, we may handle `std` just like any other instance, for example the handler

```
handler std#write _ k -> k ()
```

erases all output, while

```
let accumulate = handler
  | std#write x k -> let (v, xs) = k () in (v, x :: xs)
  | val v -> (v, [])
```

intercepts output and accumulates it in a list so that

```
with accumulate handle
  std#write "hello"; std#write "world"; 3 * 14
```

prints nothing and evaluates to `(42, ["hello"; "world"])`. Similarly, one could feed the input from a list with the handler

```
let read_from_list lst = handler
  | std#read () k -> (fun (s::lst') -> k s lst')
  | val x -> (fun _ -> x)
  | finally f -> f lst
```

Both handlers can be quite useful for unit testing of interactive programs.

6.7 Ambivalent choice and backtracking

We continue with variations of choice from Section 6.1. Recall that *ambivalent* choice is an operation which selects among several options in such a way that the overall computation succeeds. We first define the relevant types:

```
type 'a result = Failure | Success of 'a

type 'a selection = effect
  operation select : 'a list -> 'a
end
```

The handler which makes `select` ambivalent is

```
let amb s = handler
  | s#select lst k ->
    let rec try = function
      | [] -> Failure
      | x::xs -> (match k x with
                    | Failure -> try xs
                    | Success y -> Success y)
    in
      try lst
```

Given a list of choices `lst`, the handler passes each one to the continuation `k` in turn until it finds one that succeeds. The net effect is a depth-first search with which we may solve traditional problems, such as the 8 queens problem:

```
let no_attack (x,y) (x',y') =
  x <> x' && y <> y' && abs (x - x') <> abs (y - y')

let available x qs =
  filter (fun y -> forall (no_attack (x,y)) qs)
         [1;2;3;4;5;6;7;8]

let s = new selection in
with amb s handle
  let rec place x qs =
    if x = 9 then Success qs else
      let y = s#select (available x qs) in
      place (x+1) ((x,y) :: qs)
  in place 1 []
```

The function `filter` computes the sublist of those elements in a list that satisfy the given criterion. The auxiliary function `available` computes a list of available rows in column `x` if queens `qs` have been placed onto the board so far. As usual, the program places the queens onto the board by increasing column numbers: given a column `x` and the list `qs` of the queens placed so far, an available row `y` is selected for the next queen. Because the backtracking logic is contained entirely in the handler, we may easily switch from a depth-first search to a breadth-first search by replacing *only* the `amb` handler with

```
let bfs s =
  let q = ref [] in
  handler
    | s#select lst k ->
      (q := !q @ (map (fun x -> (k,x)) lst) ;
       match !q with
        | [] -> Failure
        | (k,x) :: lst -> q := lst ; k x)
```

The `bfs` handler maintains a stateful queue `q` of choice points `(k,x)` where `k` is a continuation and `x` an argument to be passed to it. The `select` operation enqueues new choice points, dequeues a choice point, and activates it.

The fact that `bfs` seamlessly combines a stateful queue with multiple activations of continuations may lure one into writing an imperative solution to the 8 queens problem such as

```
let s = new selection in
with amb s handle
  let qs = ref [] in
  for x = 1 to 8 do
    let y = s#select (available x !qs) in
    qs := (x,y) :: !qs
  done ;
  Success !qs
```

However, because `qs` is handled with a resource *outside* the scope of `amb` a queen once placed onto the board is never taken off, so the search fails. To make sure that `amb` restores the state when it backtracks, the state has to be handled *inside* its scope:

```
let s = new selection in
with amb s handle
  let qs = new ref in
  with state qs [] handle
    for x = 1 to 8 do
      let y = s#select (available x !qs) in
      qs := (x,y) :: !qs
    done ;
    Success !qs
```

The program finds the same solution as the first version. The moral of the story is that even though effects combine easily, their combinations are not always easily understood.

6.8 Selection functionals

The `amb` handler finds an answer if there is one, but provides no information on the choices it made. If we care about the choices that lead to a particular answer we proceed as follows. First we adapt the `select` operation so that it accepts a *choice point* as well as a list of values to choose from:

```
type ('a, 'b) selection = effect
```

```
  operation select: 'a * 'b list -> 'b
end
```

The idea is that we would like to record which value was selected at each choice point. Also, multiple invocations of the same choice point should all lead to the same selection. The handler which performs such a task is

```
let select s v = handler
  | s#select (x,ys) k -> (fun cs ->
    (match assoc x cs with
     | Some y -> k y cs
     | None ->
         let rec try = function
           | [] -> Failure
           | y::ys ->
               (match k y ((x,y)::cs) with
                  | Success lst -> Success lst
                  | Failure -> try ys)
         in try ys))
  | val u -> (fun cs ->
      if u = v then Success cs else Failure)
  | finally f -> f [] ;;
```

The function `assoc` performs lookup in an associative list. The handler keeps a list `cs` of choices made so far. It handles `select` by reusing a choice that was previously recorded in `cs`, if there is one, or else by trying in turn the choices `ys` until one succeeds. A value is handled as success if it is the desired one, and as a failure otherwise.

A simple illustration of the handler is a program which looks for a Pythagorean triple:

```
let s = new selection in
with select s true handle
  let a = s#select ("a", [5;6;7;8]) in
  let b = s#select ("b", [9;10;11;12]) in
  let c = s#select ("c", [13;14;15;16]) in
    a*a + b*b = c*c
```

It evaluates to `Success [("c", 13); ("b", 12); ("a", 5)]`.

Martín Escardó's "impossible" selection functional [3] may be implemented with our selection handler. Recall that the selection functional ϵ accepts a propositional function $p : 2^{\mathbb{N}} \to 2$ and outputs $x \in 2^{\mathbb{N}}$ such that $p(x) = 1$ if, and only if, there exists $y \in 2^{\mathbb{N}}$ such that $p(y) = 1$. Such an x can be found by passing to p an infinite sequence of choice points, each selecting either `false` or `true`, as follows:

```
let epsilon p =
  let s = new selection in
  let r = (with select s true handle
             p (fun n -> s#select (n, [false; true])))
  in
    match r with
      | Failure -> (fun _ -> false)
```

```
  | Success lst ->
    (fun n -> match assoc n lst with
              | None -> false | Some b -> b)
```

The `select` handler either fails, in which case it does not matter what we return, or succeeds by computing a list of choices for which `p` evaluates to `true`. In other words, `r` is a basic open neighbourhood on which `p` evaluates to `true`, and we simply return one particular function in the neighbourhood.

There are several differences between our implementation and Escardó's Haskell implementation. First, our implementation is *not* recursive, or to be more precise, it only employs structural recursion and whatever recursion is contained in `p`. Second, we compute a basic neighbourhood on which `p` evaluates to `true` and then pick a witness in it, whereas the Haskell implementation directly computes the witness. Third, and most important, we heavily use the intensional features of *eff* to direct the search, i.e., we pass a specially crafted argument to `p` which allows us to discover how `p` uses its argument. The result is a more efficient implementation of `epsilon`, which however is not extensional. A Haskell implementation must necessarily be extensional, because all total functionals in Haskell are.

6.9 Probabilistic choice

Probabilistic choice is a form of nondeterminism in which choices are made according to a probability distribution. For example, we might define an operation which picks an element from a list according to the given probabilities:

```
type 'a random = effect
  operation pick : ('a * float) list -> 'a
end
```

The operation `pick` accepts a finite probability distribution, represented as a list of pairs whose second components are nonnegative numbers that add up to 1. The handler which computes the expected value of a computation of type `float` is fairly simple

```
let expectation r = handler
  | val v -> v
  | r#pick lst k ->
      fold_right (fun (x,p) e -> e +. p *. k x) lst 0.0
```

Here `fold_right` is the list folding operation, e.g., `fold_right f [a;b;c] x` evaluates as `f a (f b (f c x))`.

Computing the distribution of results of a computation is not much more complicated:

```
let combine =
  let scale p xs = map (fun (i, x) -> (i, p *. x)) xs in
  let rec add (i,x) = function
    | [] -> [(i,x)]
    | (j,y)::lst ->
      if i = j then (j,x+.y)::lst else (j,y)::add(i,x) lst
  in
```

```
    fold_left
      (fun e (d,p) -> fold_right add (scale p d) e) []

let distribution r = handler
  | val v -> [(v, 1.0)]
  | r#pick lst k ->
      combine (map (fun (x,p) -> (k x, p)) lst)
```

Here, `combine` is the multiplication for the distribution monad that combines a distribution of distributions into a single distribution. The function `map` is the familiar one, while `fold_left` is the left-handed counterpart of `fold_right`.

As an example, let us consider the distribution of positions in a random walk of length 5, where we start at the origin, and step to the left, stay put, or step to the right with probabilities $2/10$, $3/10$ and $5/10$, respectively. The distribution is computed by

```
let r = new random in
let x = new ref in
with distribution r handle
with state x 0 handle
  for i = 1 to 5 do
    x := !x + r#pick [(-1,0.2); (0,0.3); (1,0.5)]
  done ;
  !x
```

Just like in the 8 queen example from Section 6.7 the handler for state must be enclosed by the distribution handler. We were surprised to see that the "wrong" order still works:

```
let r = new random in
let x = new ref in
with state x 0 handle
with distribution r handle
  for i = 1 to 5 do
    x := !x + r#pick [(-1,0.2); (0,0.3); (1,0.5)]
  done ;
  !x
```

How can this be? The answer is hinted at by *eff* which issues a warning about arbitrary sequencing of effects in the assignment to `x`. If we write the program with less syntactic sugar, we must decide whether to write the body of the loop as

```
let a = !x in
let b = r#pick [(-1,0.2); (0,0.3); (1,0.5)] in
  x := a + b
```

or as

```
let b = r#pick [(-1,0.2); (0,0.3); (1,0.5)] in
let a = !x in
  x := a + b
```

In the first case `a` holds the value of `x` as it is *before* probabilistic choice happens, so update correctly reinstates the value of `x`, whereas in the second case it fails to do so. Indeed, we get the wrong answer if we swap the summands in the assignment to `x` and

handle state on the outside. On one hand we should not be surprised that the order in which effects happen matters, but on the other it is unsatisfactory that a simple change in the order of addition matters so much. Perhaps the sequencing warnings should be errors after all.

6.10 Cooperative multithreading

Cooperative multithreading is a model for parallel programming in which several threads run in parallel, but only one at a time. A new thread is created with a `fork` operation, a running thread relinquishes control with a `yield` operation, and a scheduler decides which thread runs next. As is well known, cooperative multithreading can be implemented in languages with first-class continuations.

To get cooperative multithreadding in *eff* we first define an effect type with the desired operations:

```
type coop = effect
  operation yield : unit -> unit
  operation fork : (unit -> unit) -> unit
end
```

Next we define a scheduler, in our case one that uses a round-robin strategy, as a handler:

```
let round_robin c =
  let threads = ref [] in
  let enqueue t = threads := !threads @ [t] in
  let dequeue () =
    match !threads with
    | [] -> ()
    | t :: ts -> threads := ts ; t ()
  in
  let rec scheduler () = handler
    | c#yield () k -> enqueue k ; dequeue ()
    | c#fork t k ->
        enqueue k ; with scheduler () handle t ()
    | val () -> dequeue ()
  in
    scheduler ()
```

The handler keeps a queue of inactive threads, represented as thunks. Note that dequeuing automatically activates the dequeued thunk. Yield enqueues the current thread, i.e., the continuation, and activates the first thread in the queue. Fork enqueues the current thread and activates the new one (an alternative would enqueue the new thread and resume the current one). The handler must not just activate the newly forked thread but also wrap itself around it, lest `yield` and `fork` triggered by the new thread go unhandled. Thus the definition of the handler is recursive. The `val` clause makes sure that the threads in the queue get a chance to run when the current thread terminates.

Nothing prevents us from combining threads with other effects: threads may use common or private state, they may raise exceptions, inside a thread we can have another level of multithreading, etc.

6.11 Delimited control

Our last example shows how to implement standard delimited continuations in *eff*. As a result we can transcribe code that uses continuations directly into *eff*, although we have found that transcriptions are typically cleaner and easier to understand if we modify them to use operations and handlers directly.

We consider the static variant of delimited control that uses `reset` and `shift` [2]. The first operation delimits the scope of the continuation and the second one applies a function to it, from which it follows that one acts as a handler and the other as an operation. Indeed, the *eff* implementation is as follows:

```
type ('a, 'b) delimited =
effect
  operation shift : (('a -> 'b) -> 'b) -> 'a
end

let rec reset d = handler
  | d#shift f k -> with reset d handle (f k)
```

Since `f` itself may call `shift`, the handler wraps itself around `f k`. The standard useless example of delimited control is

```
let d = new delimited in
with reset d handle
  d#shift (fun k -> k (k (k 7))) * 2 + 1
```

The captured continuation `k` multiplies the result by two and adds one, thus the result is $2 \times (2 \times (2 \times 7 + 1) + 1) + 1 = 63$. In Scheme the obligatory example of (undelimited) continuations is the "yin yang puzzle", whose translation in *eff* is

```
let y = new delimited in
with reset y handle
  let yin  =
    (fun k -> std#write "@" ; k) (y#shift (fun k -> k k))
  and yang =
    (fun k -> std#write "*" ; k) (y#shift (fun k -> k k))
  in
    yin yang
```

The self-application `k k` is suspect, and *eff* indeed complains that it cannot solve the recursive type equation $\alpha = \alpha \to \beta$. We have not implemented unrestricted recursive types, but we can turn off type checking, after which the puzzle prints out the same answer as the original one in Scheme.

7 Discussion

Our purpose was to design a programming language based on the algebraic approach to computational effects and their handlers. We feel that we succeeded and that our experiment holds many promises.

First, we already pointed out several times that *eff* would benefit from an effect system that provided a static analysis of computational effects. However, for a useful result we need to find a good balance between expressivity and complexity.

Next, it is worth investigating how to best reason about programs in *eff*. Because the language has been inspired by an algebraic point of view, it seems clear that we should look into equational reasoning. The general theory has been investigated in some detail [15, 17], but the addition of effect instances may complicate matters.

Finally, continuations are the canonical example of a non-algebraic computational effect, so it is a bit surprising that *eff* provides a flexible and clean form of delimited control, especially since continuations were not at all on our design agenda. What then can we learn from *eff* about control operators in an effectful setting?

References

[1] 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.

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

[3] Martín Escardó and Paulo Oliva. Selection functions, bar recursion and backward induction. *Mathematical Structures in Computer Science*, 20:127–168, 2010.

[4] Martin Hyland, Paul Blain Levy, Gordon Plotkin, and John Power. Combining algebraic effects with continuations. *Theoretical Computer Science*, 375(1–3):20–40, 2007.

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

[6] 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.

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

[8] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[10] 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.

[11] 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.

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

[13] Gordon Plotkin and John Power. Tensors of comodels and models for operational semantics. In Andrej Bauer and Michael Mislove, editors, *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV)*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 295–311, 2008.

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

[15] Gordon David Plotkin and Matija Pretnar. A logic for algebraic effects. In *23rd Symposium on Logic in Computer Science*, pages 118–129, 2008.

[16] John Power and Olha Shkaravska. From comodels to coalgebras: State and arrays. *Electronic Notes in Theoretical Computer Science*, 106:297–314, 2004.

[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] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[20] Andrew Wright. Simple imperative polymorphism. In *LISP and Symbolic Computation*, pages 343–356, 1995.