Eff Directly in OCaml

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

KC Sivaramakrishnan
University of Cambridge, UK
sk826@cam.ac.uk

Abstract
We present the embedding of the language Eff into OCaml, using the library of delimited continuations or the OCaml-effects branch. The embedding is systematic, lightweight, performant and supports even higher-order, ‘dynamic’ effects with their polymorphism. OCaml thus may be regarded as another implementation of Eff, broadening the scope and appeal of that language.

1. Summary
The Eff language\(^1\) is an OCaml-like language centered on algebraic effects\(^1\), designed to try out algebraic effects on a larger scale and gain practical experience using them. It is currently implemented as an interpreter, with the compiler to OCaml in the works.

Rather than compile Eff to OCaml, we embed it: the parts of the language that do not use effects are the OCaml itself, with no translations or alterations. The Eff-specific parts are translated to OCaml that uses the library of delimited control delimcc\(^3\) or the new effects of the OCaml-effects branch\(^2\). The translation is local and straightforward. In fact, it is so simple that it is currently done by hand. We thus present the set of OCaml idioms for effectful programming with the almost exact look-and-feel of Eff.

We idiomatically support even the ‘dynamic effects’ of Eff 3.1 with their attendant polymorphism, offering the more general view: on our translation, the dynamic creation of effects is but another effect, with no special syntax or semantics.

The source code of all our examples and benchmarks is available at http://okmij.org/ftp/continuations/Eff/.

2. Eff in Itself and OCaml
We illustrate the Eff embedding on the running example, juxtaposing Eff code with the corresponding OCaml. We thus demonstrate both the simplicity of the translation and the way to do Eff-like effects in the idiomatic OCaml.

An effect in Eff has to be declared first:
\[
\text{type } \alpha \text{ nondet } = \text{ effect}\n\]
operation fail : unit \to empty
operation choose : (\alpha \to unit) \to unit

Our running effect is thus the familiar non-determinism. The declaration introduces only the names of effect operations and their types. The semantics is to be defined by a handler later on.

In OCaml, the Eff declaration is rendered as the data type:
\[
\text{type } \alpha \text{ nondet } = \n\]
| Fail of unit \to unit
| Choose of \alpha \to unit

assuming the previously defined types
\[
\text{type empty} \n\]
\[
\text{type } \epsilon \text{ result } = \text{ Done } | \text{ Eff of } \epsilon
\]

The translation pattern should be easy to see: each data type variant has exactly two arguments, the latter is the continuation. The attentive reader quickly recognizes the freer monad\(^4\).

Next we “instantiate the effect signature”, as Eff puts it:
\[
\text{let } r = \text{ new nondet } \n\]
after which we can write the sample Eff code with the non-determinism effect:
\[
\text{let } f () = \n\]
\[
\text{let } x = r\#\text{choose } (\text{"a"}, \text{"b"}) \text{ in} \n\]
\[
\text{print_string } x ; \n\]
\[
\text{let } y = r\#\text{choose } (\text{"c"}, \text{"d"}) \text{ in} \n\]
\[
\text{print_string } y \n\]

In OCaml, the effect instantiation takes the form
\[
\text{let } r = \text{ new_prompt } (\n\]
calling the function from the delimcc library. An effect instance of Eff hence corresponds to a prompt of delimcc. We are now ready to translate the sample Eff code to OCaml. The translation looks cleaner if we first define helper functions, for each effect of the nondet signature:
\[
\text{let choose : } \alpha \text{ nondet result prompt } \to \alpha \to \alpha = \text{ fun p arg } \to \n\]
\[
\text{shift0 } p (\text{fun k } \to \text{Eff } (\text{Choose } (\text{arg,k}))) \n\]

and similar for fail. These definitions are entirely regular and can be mechanically generated. The (inferred) signature tells that OCaml’s choose is quite like Eff’s r#choose: it takes the effect instance (prompt) and a pair of values and (non-deterministically) returns one of them. Strictly speaking, however, choose does hardly anything: it merely captures the continuation and packs it, along with the argument in the data structure, to be passed to the effect handler. It is the handler that does the choosing.

The sample Eff code can be literally pasted into OCaml, with small stylistic adjustments:
\[
\text{let } f () = \n\]
\[
\text{let } x = \text{ choose } r (\text{"a"}, \text{"b"}) \text{ in} \n\]
\[
\text{printf } \text{"x is }%s\text{" }\text{in } x ; \n\]
\[
\text{let } y = \text{ choose } r (\text{"c"}, \text{"d"}) \text{ in} \n\]
\[
\text{printf } \text{"y is }%s\text{" }\text{in } y ; \n\]

The effect instance r is passed to choose as the regular argument, without any special r# syntax.

To run the sample code, we have to tell how to interpret the effect actions Choose and Fail: so far, we have only defined their names and types — the algebraic signature. It is the interpreter of the actions, the handler, that infuses the action operations with their meaning. For example, Eff may execute the sample code by interpreting choose to follow up on both choices, depth-first:
\[
\text{let test}1 = \text{handle f () with} \n\]
\[
| \text{val x } \to \text{x} ; \n\]
\[
| r\#\text{choose } (x, y) \to k \to k x ; k y \n\]
\[
| r\#\text{fail } () \to () \n\]

\(^2\)The connection to the freer monad points out that \(\alpha \text{ nondet} \) does not really need the parameter — neither in OCaml, nor, more importantly, in Eff.

\(^1\)http://www.eff-lang.org/
Having defined \( \text{get} \) and \( \text{put} \) effect-sending functions like choose before, we can use state as we did nondet:

\[
\begin{align*}
\text{let} \quad & a = \text{new\_prompt} () \ \text{in} \\
& \text{handle\_it} \ a \ (\text{fun} \ r \ \rightarrow) \\
& u = \text{get} a () \ \text{in} \ \text{let} \ \nu = \text{get} a () \ \text{in} \\
& \text{put} a (\nu + 30); \ \text{let} \ \omega = \text{get} a () \ \text{in} \ ((\omega, \nu)) \\
& (\text{handler\_ref} 10)
\end{align*}
\]

The handler in Eff (and in OCaml) is a function and so can be detached (defined separately) as we have just done for the handler of state requests. It receives as argument the initial state value.

\[
\begin{align*}
\text{let} \quad & \text{rec} \ \text{handler\_ref} \ s \res = \text{function} \\
& \text{Done} \rightarrow \ \text{get\_result} \ res \\
& \text{Eff\ Get} (\_ , k) \rightarrow \ \text{handler\_ref} \ s \res \odot k \ s \\
& \text{Eff\ Put} (s , k) \rightarrow \ \text{handler\_ref} \ s \res \odot k (k)
\end{align*}
\]

To really treat an instance of state as a reference cell, we need a way to create many state effects of many types. Whenever we need a new reference cell, we should be able to create a new instance of the state signature \( \text{and} \) to wrap the program with the handler for the just created instance. The first part is easy, especially in the OCaml embedding: the effect-instance-creating \( \text{new\_prompt} \) is the ordinary function, and hence can be called anywhere and many times. To just as dynamically wrap the program in the \( \text{handle\_it} \ldots (\text{handler\_ref} \ n) \) block is complicated. Eff had to introduce ‘default handlers’ for a signature instance, with a special syntax and semantics. An effect not handled by an ordinary (local) handler is given to the default handler, if any.

Our OCaml demonstration that dynamic effects require nothing special: Creating a new instance and handling it may be treated as an ordinary effect:

\[
\begin{align*}
\text{type} \quad & \epsilon \ \text{handler\_t} = (h: \forall \omega. \ \omega \ \text{result\_value} \rightarrow \epsilon \ \text{result} \rightarrow \omega) \\
\text{type} \quad & \text{dyn\_instance} = \text{New}: \epsilon \ \text{handler\_t} \ \epsilon (\text{result\_prompt} \rightarrow \text{dyn\_instance\_result} \rightarrow \text{dyn\_instance}) \\
\text{let} \quad & \text{new\_instance} \ p \ \text{arg} = \text{shift0} p (\text{fun} k \rightarrow \text{Eff} (\text{New} (\text{arg}, k)))
\end{align*}
\]

The New effect receives as the argument the handling function \( h \). The New handler creates a new instance \( p \) and passes it as the reply to the continuation – at the same time wrapping the continuation into the handling block \( \text{handle\_it} \ldots h \):

\[
\begin{align*}
\text{let} \quad & \text{rec} \ \text{new\_handler\_res} = \text{function} \\
& \text{Done} \rightarrow \ \text{get\_result} \ res \\
& \text{Eff\ New} (h = (h, k)) \rightarrow \\
& \text{let} \ p = \text{new\_prompt} () \ \text{in} \\
& \ \text{handle\_it} p \ (\text{fun} \ o \ \rightarrow \ \text{new\_handler\_res} \odot k \ p \ o) h
\end{align*}
\]

Both steps of the dynamic effect creation are hence accomplished by the ordinary handler. The allocation of a reference cell is hence

\[
\begin{align*}
\text{let} \quad & \text{new\_ref} s0 = \text{new\_instance} \ \text{pnew} \s h = \text{handler\_ref} s0 \\
& \text{val} \ \text{new\_ref} : \alpha \rightarrow \text{state\_result\_prompt} = <\text{fun}> \\
& \text{Since} \ \text{new\_ref} \ \text{is polymorphic, it may allocate cells of arbitrary types.}
\end{align*}
\]

The New effect, albeit 'higher-order', is not special. Programmers may write their own handlers for it (and for the state, for that matter), to implement, e.g., transactional state.

In conclusion, we have demonstrated the embedding of Eff 3.1 in OCaml by a simple, local translation. It allows to almost cut-and-paste Eff code into OCaml, with simple adjustments. Theoretically, the framework of delimited continuation has clarified the thorny dynamic effects, demonstrating that there is nothing special about them. Dynamic effect creation can be treated as an ordinary effect. The accompanying code shows several examples, including the queens benchmark.

### References