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 a compiler to OCaml in the works.

Rather than compile Eff to OCaml, we *embed* it. After all, save for algebraic effects, Eff truly is a subset of OCaml. The effect-specific parts are translated to the OCaml code 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 a set of OCaml idioms for effectful programming with the almost exact look-and-feel of Eff.

We idiomatically support even 'dynamic effects' of Eff 3.1 with their attendant polymorphism, offering a 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 idiomatic OCaml.

An effect in Eff has to be declared first[2]:

```
type α nondet = effect
  operation fail   : unit → empty
  operation choose : (α ∗ α) → α
end
```

Our running effect is thus 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:

```
type α nondet =
  | Fail   of unit    ∗ (empty → α nondet result)
  | Choose of (α ∗ α) ∗ (α     → α nondet result)
```

assuming the previously defined types

```
type empty
type ϵ result = Done | Eff of ϵ
```

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][3].

Next we "instantiate the effect signature", as Eff puts it:

```
let r = new nondet
```

after which we can write the sample Eff code with the non-determinism effect:

```
let f () =
  let x = r#choose ("a", "b") in
   print_string x ;
  let y = r#choose ("c", "d") in
   print_string y
```

In OCaml, the effect instantiation takes the form

```
let r = new_prompt ()
```

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:

```
let choose : α nondet result prompt → α ∗ α → α = fun p arg →
  shift0 p (fun k → Eff (Choose (arg,k)))
```

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:

```
let f () =
  let x = choose r ("a","b") in
   print_string x;
  let y = choose r ("c","d") in
   print_string y;
```

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:

```
let test1 = handle f () with
  | val x → x
  | r#choose (x, y) k → k x ; k y
  | r#fail () _ → ()
```

[1] `http://www.eff-lang.org/`

[2] Eff code is marked with double lines to distinguish it from OCaml.

[3] The connection to the freer monad points out that α nondet does not really need the parameter – neither in OCaml, nor, more importantly, in Eff.

That is, when asked to choose, we continue the program with the first alternative – and continue it again, with the second one.

The Eff handling can be performed in OCaml in the same way, having built a small bit of infrastructure. Recall, the result of every ϵ action has the type ϵ result, which does not include the type of the result. The result has to be extracted via a side-effect then:

```
type α result_value = α option ref
let get_result : α result_value → α = fun r →
  match !r with Some x → r := None; x
```

This round-about extraction trick gives "answer-type" polymorphism without the first-class polymorphism and the attendant awkwardness. The other bit of the infrastructure is the boilerplate to set the prompt (limiting the extent of the continuation captured by the effect action) and to save the result, to be extracted by the handler:

```
let handle_it :
    α result prompt →                    (* effect instance *)
    (unit → ω) →                         (* expression *)
    (ω result_value → α result → γ) → (* handler *)
    γ = fun effectp exp handler →
  let res = ref None in
  handler res (push_prompt effectp @@ fun () →
    res := Some (exp ()); Done)
```

The Eff handler example looks in OCaml as:

```
let test1 = handle_it r f @@ fun res →
  let rec handler = function
    | Done → get_result res
    | Eff Choose ((x,y),k) → handler (k x); handler (k y)
    | Eff Fail (x,k) → ()
  in handler
```

The only difference is that Eff handlers are "deep" (that is, they automatically re-apply to the continued computations) whereas our handlers are shallow, and we have to re-apply them manually. Other than that, the translation is straightforward and local.

Eff supports nested handlers; e.g., the inner nondet handler may handle only Choose in some special way, leaving Fail for the outer handler. The inner handler may also do its own nondet effects, to be dealt with by the outer handler then. All these cases are translated to OCaml in the manner just outlined, and just as straightforwardly. The accompanying code shows several examples.

3. Higher-order Effects

Our running example used the single instance r of the nondet effect, created at the top level – essentially, 'statically'. Eff also supports creating effect instances as the program runs. These, 'dynamic effects' let us, for example, implement reference cells as instances of the state effect. The realization of this elegant idea required extending Eff with default handlers, with their special syntax and semantics. The complexity was the reason dynamic effects were removed from Eff 4.0 (but may be coming back).

The OCaml embedding of Eff gave us the vantage point of view to realize that dynamic effects may be treated themselves as an effect. This New effect may create arbitrarily many instances of arbitrary effects of arbitrary types. Below we briefly describe the challenge of dynamic effects and its resolution in OCaml.

We take the state effect as the new running example:

```
type α state =
  | Get of unit * (α → α state result)
  | Put of α * (unit → α state result)
```

Having defined get and put effect-sending functions like choose before, we can use state as we did nondet:

```
let a = new_prompt () in
handle_it a (fun () →
  let u = get a () in let v = get a () in
  put a (v + 30); let w = get a () in (u,v,w))
 (handler_ref 10)
```

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.

```
let rec handler_ref s res = function
  | Done → get_result res
  | Eff Get (_,k) → handler_ref s res @@ k s
  | Eff Put (s,k) → handler_ref s res @@ k ()
```

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 *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 new_prompt is the ordinary function, and hence can be called anywhere and many times. To just as dynamically wrap the program in the handle_it ... (handler_ref n) block is complicated. Eff had to introduce 'default handlers' for a signature instance, with special syntax and semantics. An effect not handled by an ordinary (local) handler is given to the default handler, if any.

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

```
type ϵ handler_t = {h: ∀ω. ω result_value → ϵ result → ω}
type dyn_instance =
    New : ϵ handler_t * (ϵ result prompt → dyn_instance result )
      → dyn_instance
let new_instance p arg = shift0 p (fun k → Eff (New (arg,k)))
```

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 handle_it ... h:

```
let rec new_handler res = function
  | Done → get_result res
  | Eff New ({h= h},k) →
      let p = new_prompt () in
      handle_it p (fun () → new_handler res @@ k p) h
```

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

```
let pnew = new_prompt ()
let newref s0 = new_instance pnew {h = handler_ref s0}
⇝    val newref : α → α state result prompt = <fun>
```

Being polymorphic, newref may allocate cells of arbitrary types.

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

In conclusion, we have demonstrated the embedding of Eff 3.1 in OCaml by a simple, local translation. We may 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

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. .

[2] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects, 2015. OCaml Users and Developers Workshop.

[3] O. Kiselyov. Delimited control in OCaml, abstractly and concretely. *Theoretical Computer Science*, 435:56–76, June 2012. .

[4] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Haskell*, pages 94–105. ACM, 2015. .