Eff Directly in OCaml

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

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

The language Eff is an OCaml-like language serving as a prototype implementation of the theory of algebraic effects, intended for experimentation with algebraic effects on a large scale.

We present the embedding of Eff into OCaml, using the library of delimited continuations or the multicore OCaml branch. We demonstrate the correctness of the embedding denotationally, relying on the tagless-final–style executable denotational semantics, including the novel denotational semantics of delimited control. 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 Introduction

Algebraic effects [19, 18] are becoming more and more popular approach for expressing and composing computational effects. There are implementations of algebraic effects in Haskell [10, 12], Idris [3], OCaml [5], Koka [14], Scala, Javascript[1], PureSctipt[2], and other languages. The most direct embodiment of algebraic effect theory is the language Eff[3] "built to test the mathematical ideas of algebraic effects in practice". It is an OCaml-like language with the native facilities (syntax, type system) to declare effects and their handlers [2]. It is currently implemented as an interpreter, with an optimizing 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 [11] or the new effects of the Multicore OCaml branch [5]. The translation is local and straightforward. We thus present a set of OCaml idioms for effectful programming with the almost exact look-and-feel of Eff.

Our second contribution is the realization that so-called 'dynamic effects', or handled 'resources', of Eff 3.1 (epitomized by familiar reference cells, which can be created in any number and hold values of any type) is not a separate language feature. Rather, the dynamic creation of effects is but another effect, and hence is already supported by our implementation of ordinary effects and requires no special syntax or semantics.

As a side contribution, we show the correctness of our embedding of Eff in OCaml denotationally, relying on the "tagless-final" style of executable denotational semantics. We also demonstrate the novel, denotational semantics of delimeted control.

The structure of the paper is as follows. First we informally introduce Eff on a simple example. §2.2 then demonstrates our translation to OCaml using the delimcc library, putting Eff and the corresponding OCaml code side-by-side. §2.3 shows how the embedding works in multicore OCaml with its 'native

[1]https://www.humblespark.com/blog/extensible-effects-in-node-part-1
[2]http://purescript.org
[3]http://www.eff-lang.org/

effects'. §3 gives the formal, denotational treatment, reminding the denotation semantics of Eff; describing the novel denotational semantics of delimited control; then presenting the translation precisely; and arguing that it is meaning-preserving. §5 evaluates the performance of our implementation of Eff comparing it with the Eff's own optimizing compiler. We describe the translation of the dynamic effects into OCaml in §4. Finally, we conclude and summarize the research program inspired by our Eff embedding.

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 the 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.

2.1 A taste of Eff

An effect in Eff has to be declared first[4]:

```
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 – the failure and the non-deterministic choice between two alternatives – and their types. The semantics is to be defined by a handler later on. All effect invocations uniformly take an argument (even if it is dummy ()) and promise to produce a value (even if of the type empty, of which no values exists; the program hence cannot continue after a failure). The declaration is parameterized by the type α of the values to non-deterministically choose from. (Using first-class polymorphism of OCaml, we can avoid the parameterization. Rather, we may give choose the polymorhic signature forall $\alpha.\ \alpha * \alpha \to \alpha$.)

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

```
let r = new nondet
```

One may think of an instance r as part of the name for effect operations: the signature nondet defines the common interface. Different part of a program may independently use non-determinism if each creates an instance for its own use. Unlike the effect declaration, which is static, one may create arbitrarily many instances at run-time.

We can now write the sample non-deterministic Eff code:

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

The computation (using the Eff terminology [2]) r#choose ("a", "b") invokes the effect choose on instance r, passing the pair of strings "a" and "b" as parameters. Indeed the instance feels like a part of the name for an effect operation. The name of the effect hints that we wish to choose a string from the

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

pair. Strictly speaking however, choose does not have any meaning beyond signalling the intention of performing the 'choose' effect, whatever it may mean, on the pair of strings.

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 () _       → ()
```

The handle form is deliberately made to look like the **try** form of OCaml – following the principle that algebraic effects are a generalization of ordinary exceptions. The fail action is treated as a genuine exception: if the computation f () invokes fail (), test_1 immediately returns with (). When the computation f () terminates with a value, the **val** x branch of the handle form is evaluated, with x bound to that value; test_1 hence returns the result of f () as it is[5]. The choose action is the proper effect: when the evaluation of f () comes across r#choose ("a","b"), the evaluation is suspended and the r#choose clause of the handle form above is executed, with x bound to "a", y bound to "b" and k bound to the continuation of f () evaluation, up to the handle. Since k is the delimited continuation, it acts as a function, returning what the entire handle form would return (again unit, in our case). Thus the semantics given to r#choose in test_1 is first to choose the first component of the pair; and after the computation with the choice is completed, choose the second component. The choose effect hence acts as a *resumable* exception, familiar from Lisp. In our case, it is in fact resumed twice. Executing test_1 prints acdbcd.

Just like the **try** forms, the handlers may nest: and here things become more interesting. First of all, distinct effects – or distinct instances of the same effect – act independently, unaware of each other. For example, it is rather straightworward to see that the following code (where the i1 and i2 handlers make the choice slighly differently)

```
let test2 =
 let i1 = new nondet in
 let i2 = new nondet in
 handle
   handle
     let x = i1#choose ("a", "b") in
      print_string x ;
     let y = i2#choose ("c", "d") in
      print_string y
   with
   | val () → print_string ";"
   | i2#choose (x,y) k → k x; k y
 with
 | val x → x
 | i1#choose (x,y) k  → k y; k x
```

prints bc;d;ac;d; The reader may try to work out the result when the inner handler handles the i1 instance and the outer one i2.

One may nest handle forms even for the same effect instance. To confidently predict the behavior in that case one really needs the formal semantics, overviewed in §3.1. First, the effect handling code may itself invoke effects, including the very same effect:

[5] An astute reader must have noticed that this result must again be unit.

```
let testn1 =
 handle
  handle
   let x = r#choose ("a", "b") in
    print_string x
  with
   | val () → print_string ";"
   | r#choose (x,y) k → k (r#choose(x,y))
 with
  | val x → x
  | r#choose (x,y) k → k y; k x
```

The effect "re-raised" by the inner handler is then dealt with by an outer handler. In testn1 hence the inner handler simply relays the choose action to the outer one. The code prints b;d;

A handler does not have to handle all actions of a signature. The unhandled ones are quietly "re-raised" (again, similar to ordinary exceptions):

```
let testn2 =
 handle
  handle
   let x = r#choose ("a", "b") in
    print_string x;
   (match r#fail () with)
  with
   | val ()        → print_string ";"
   | r#fail () _ → print_string "!"
 with
  | val x → x
  | r#choose (x,y) k → k y; k x
;;
```

The code prints b!a!. The main computation does both fail and choose effects; the inner handler deals only with fail, letting choose propagate to the outer one. An unhandled effect action is a run-time error. The suspicious (**match** r#fail () **with**) expression does a case analysis on the empty type. There are no values of that type and hence no cases are needed.

2.2 Eff in OCaml

We now demonstrate how the Eff examples from the previous section can be represented in OCaml, using the library of delimited control delimcc [11]. We intentionally write the OCaml code to look very similar to Eff, hence showing off the Eff idioms and introducing the translation from Eff to OCaml intuitively. We make the translation formal in §3.

Before we begin, we declare two OCaml types:

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

The abstract type empty is meant to represent the empty type of Eff, the type with no values[6]. The result type represents the domain of results R from [2, §4], to be described in more detail in §3.

We now begin with our translation, juxtaposing Eff code with the corresponding OCaml. Recall, an effect has to be declared first[7]:

[6]The fact that empty has no constructors does not mean it cannot have any: after all, the type is abstract. Defining truly an empty type in OCaml is quite a challenge, which will takes us too much into the OCaml specifics.

[7]Again, the Eff code is marked with double lines to distinguish it from OCaml.

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

In OCaml, an Eff declaration is rendered as a data type declaration:

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

that likewise defines the names of effect operations, the types of their arguments and the type of the result after invoking the effect. 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 [12][8].

To make the translation even closer correspond to Eff, we define two functions choose and fail, using the delimited control operator shift0 provided by the delimcc library.

```
let choose p arg = shift0 p (fun k → Eff (Choose (arg,k)))
(* val choose : α nondet result Delimcc.prompt → α * α → α = <fun> *)
let fail  p arg  = shift0 p (fun k → Eff (Fail (arg,k)))
(* val fail   : α nondet result Delimcc.prompt → unit → empty = <fun> *)
```

The inferred type of these functions is shown in the comments. The first argument p is a so-called prompt [11], the control delimiter. The operator shift0 p captures the continuation up to the dynamically closest occurrence of push_prompt p operation, for the same value of p. We describe the semantics of shift0 in §3.2; for now we observe that the fail and choose definitions look entirely regular and could have been mechanically generated. The inferred types looks almost like the types of the corresponding Eff operations. For example, our 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 (just like r#choose in Eff) 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 "instantiation of the effect signature"

```
let r = new nondet
```

looks into OCaml as creating a new prompt

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

whose type, inferred from the use in the code below, is α nondet result prompt. The type does look like the type of an 'instance' of the nondet effect. The created prompt can be passed as the first argument to the choose and fail functions introduced earlier.

We can now translate the sample Eff code that uses non-determinism

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

into OCaml as

[8]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.

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

The translation is almost literally cut-and-paste, with small stylistic adjustments. The effect instance r is passed to choose as the regular argument, without any special r# syntax.

To run our sample Eff code or its OCaml translation we have to define how to interpret the choose effects. In Eff, it was the job of the handler. Recall:

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

The handler has two distinct parts: one defining the interpretation of the result of f () execution (the val x clause); the rest deals with interpreting effect operations and resuming the computation interrupted by these effects. The form of the handler expression almost makes it look as if a computation such as f () may end in two distinct ways: normally, yielding a value, or by performing an effect operation. In the latter case, the result collects the arguments passed to the effect operation plus the continuation to resume the computation after the effect is handled. The denotational semantics of Eff presented in [2, §4] and reminded in §3.1 gives computations exactly such a denotation: a terminating computation is either a value or an effect operation with its arguments and the continuation. Our translation of Eff to OCaml takes such denotation to heart, representing by the ε result type.

At first glance, the result type, reminded below,

```
type ε result = Val | Eff of ε
```

seems lacking: although it indeed may carry the data about the effect operation (the argument of the Eff constructor), there is no space for the normal result of a computation: Val has no argument and carries no data. This is because we decided to carry the normal result of the computation 'out of band', to avoid indexing result by the result type of the computation. Although it is natural to give another type parameter to result, recall that it appears in the declaration of the effect data type (see α nondet) and in the prompt. The effect data type should not depend on the type of the result of the overall computation that uses the effect. Our 'out of band' carrying of the normal computation result is a cheap way to implement what is called answer-type polymorphism [1], without resorting to first-class polymorphism and the attendant awkwardness. The earlier occurrence of a similar trick is in [13, §5.2], which explains the need for the polymorphism in more detail.

The out-of-band carrying of the computation result is implemented through a reference cell:

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

One is reminded of a similar trick of extracting the result of a computation in the continuation-passing style[9] which is often used in implementations of delimited control (for example, [11])[10]. The refence cell α result_value is allocated and stored into in the following code

[9] If the continuation is given the type $\alpha \to$ empty then the often heard 'pass the identity continuation' is type-incorrect.

[10] We could have also used a related trick: exceptions.

```
let handle_it :
    α result prompt →                  (* effect instance *)
    (unit → ω) →                       (* expression *)
    (ω → γ) →                          (* val clause *)
    ((α result → γ) → α → γ) →         (* oper clause *)
    γ =
  fun effectp exp valh oph →
  let res = ref None in
  let rec loop = function
    | Val     → valh (get_result res)
    | Eff eff → oph loop eff
  in loop @@ push_prompt effectp @@ fun () → (res := Some (exp ()); Val)
```

The expression to handle (given as a thunk exp) is run after setting the prompt to delimit continuations captured by effect operations (more precisely, by shift0 underlying choose and other effect operations). If the computation finishes, the value is stored, for a brief moment, in the reference cell res, and then extracted and passed to the normal termination handler valh. Seeing how handle_it is actually used may answer the remaining questions about it:

```
let test1 = handle_it r f
  (fun x → x) @@ fun loop → function
    | Choose ((x,y),k) → loop (k x); loop (k y)
    | Fail ((),_)      → ()
```

The OCaml version of test_1 ends up very close to the Eff version. We can see that handle_it receives the 'effect instance' (the prompt r), the thunk of the computation to perform f and two handlers, for the normal termination result (which is the identity in our case) and for handling the α nondet operations, Choose and Fail. The only notable distinction from Eff is the loop argument that is composed with the effect continuation k. That composition is also happens in Eff (as we can see in §3.1), but behind the scenes: Eff hides the composition in the syntactic sugar for the handle form.

The just outlined translation applies to the nested handlers as is. For example, the test_2 code from §2.1 is translated into OCaml as follows:

```
let test2 =
 let i1 = new_prompt () in
 let i2 = new_prompt () in
 handle_it i1 (fun () →
   handle_it i2 (fun () →
     let x = choose i1 ("a", "b") in
      print_string x ;
     let y = choose i2 ("c", "d") in
      print_string y)
   (fun () → print_string ";") @@ fun loop → function
   | Choose ((x,y),k) → loop (k x); loop (k y)
 )
 (fun x → x) @@ fun loop → function
 | Choose ((x,y),k)  → loop (k y); loop (k x)
```

The translation was done by cutting-and-pasting of the Eff code and doing a few slight modifications. The code runs and prints the same result as the original Eff code. The other nested handling examples, testn1 and testn2 of §2.1 are translated in the manner just outlined, and just as straightforwardly. We refer to the source code for details.

2.3 Eff in multicore OCaml

In this section, we describe the embedding of Eff in multicore OCaml. But first we briefly describe the implementation of algebraic effects and handlers in multicore OCaml.

2.3.1 Algebraic effects in multicore OCaml

Multicore OCaml [16] is an extension of OCaml with native support for concurrency and parallelism. Concurrency in multicore OCaml is expressed through algebraic effects and their handlers. We might declare the non-determinism operations as:

```
effect Fail   : empty
effect Choose : (α ∗ α) → α
```

Unlike Eff, multicore OCaml does not provide the facility to define new effect types. Indeed, the above declarations are simply syntactic sugar for extending the in-built effect type with new operations:

```
type _ eff += Fail   : empty eff
type _ eff += Choose : α ∗ α → α eff
```

Now, we can write the original non-deterministic program as:

```
let f () =
  let x = perform (Choose ("a","b")) in
   print_string x;
  let y = perform (Choose ("c";"d")) in
   print_string y
in
match f () with
| x → x (∗ value clause ∗)
| effect Choose(x,_) k → continue k x
| effect Fail _ → ()
```

Effects are performed with the perform keyword. Multicore OCaml extends OCaml's pattern matching syntax to double up as handlers when effect patterns (patterns that begin with the keyword effect) are present. This example always chooses the first component of the pair. Continuations are resumed with continue keyword. The handlers in OCaml are deep and the handlers wrap around the continuation. Unlike Eff, algebraic effects in multicore OCaml are unchecked. Just like ambient effects in OCaml, user-defined effects in multicore OCaml have no type-level marker that decorates function types with effects performed. An effect that is not handled by any handler in the current stack raises a runtime exception.

Since the primary motivation for adding algebraic effects to OCaml is to support concurrency, by default, the continuations are one-shot and can be resumed at-most once. This restriction is enforced with dynamic checks, which raise exception when a continuation is resumed more than once. Pleasantly, this restriction allows multicore OCaml to implement the continuations in *direct-style*, by creating a new heap-managed stack object for effect handlers. Continuation capture is also cheap; capturing a continuation only involves obtaining a reference to the underlying stack object. Since the continuations are one-shot, there is no need for copying the continuation object when resuming the continuation. For OCaml, this direct-style continuations are faster than CPS translating the entire code base. This is because CPS translating the entire program allocates a lot of intermediate closures, which OCaml does not aggressively optimize. The direct-style implementation permits performance backwards compatibility; only the code that uses continuations pays the cost of creating and managing continuations. The rest of the code behaves similar to vanilla OCaml.

Multicore OCaml does include support for multi-shot continuations, by allowing the programmer to clone the continuation on-demand. Thus, the example test_1 is implemented in multicore OCaml as,

```
match f () with
| x → x (* value clause *)
| effect Choose(x,y) k →
    continue (Obj.clone_continuation k) x;
    continue k y
| effect Fail _ → ()
```

In the above, we clone the continuation k using Obj.clone_continuation, resume the continuation with x before resuming with y. Cloning deep copies the continuation, allowing the same continuation to be resumed more than once.

2.3.2 Delimcc in multicore OCaml

We now discuss the Eff embedding in multicore OCaml. We achieve the embedding by embedding the delimcc operators new_prompt, push_prompt, and shift0 in multicore OCaml. The embedding is given in Fig. 1. The prompt type is a record with two operations, one to take a sub-continuation and the other to push a new prompt. We instantiate a new prompt by declaring a new effect called Prompt in a local module. Thus, we get a new Prompt effect instance for every invocation of new_prompt. The take operation wraps the given function f in the effect constructor and performs it. push operation evaluates f in a handler which handles the Prompt effect. This handler applies the continuation to the given function f.

Now, push_prompt and take_subcont operations are simply the definitions of push and take, respectively. push_subcont unconditionally clones the continuation and resumes it. Cloning is necessary here since delimcc continuations are multi-shot. Finally, shift0 is implemented in terms of the operations to take and push continuations, following its standard definition [6, 11] (see also §3.2 for a reminder). Thus, we have embedded a subset of Delimcc operators used in the Eff embedding in multicore OCaml. And in turn, we have an embedding of Eff in multicore OCaml.

3 Eff in OCaml, Formally

In this section we formally state our translation from Eff to OCaml and argue that is meaning-preserving. First we recall the denotational semantics of Eff. §3.2 outlines the (novel) denotational semantics of delimited control, using the same semantic domains introduced in §3.1 for Eff. Finally, §3.3 defines the translation from the Core Eff to the Core OCaml plus delimited control, and argues it preserves the denotation of expressions.

3.1 The Semantics of Eff

The Eff paper [2] has introduced the language also formally, by specifying its denotational semantics. We recall it in this section for ease of reference, making small notational adjustments for consistency with the formalization of delimited control in the later section. For ease of formalization and undertstanding, we simplify the language to its bare minimum, the Core Eff, presented in Fig. 2.

Whereas Eff, as a practical language, has a number of syntactic forms, we limit the Core Eff to the basic abstractions, applications and let-expressions. Likewise, Core Eff, besides the effect types has only unit, integer and function types. Other basic types, as well as products and sums present in the full Eff are

```
module type Delimcc  sig
  type α prompt

  val  new_prompt   : unit  → α prompt
  val  push_prompt  : α prompt → (unit → α) → α
  val  shift0       : α prompt → ((β → α) → α) → β
end

module Delimcc : Delimcc =  struct
  type α prompt =  {
          take   : β. ((β,  α) continuation  → α) → β;
          push   : (unit  → α) → α;
  }

  let  new_prompt (type a) ()  :  a  prompt =
     let  module M =  struct effect Prompt : ((β,a) continuation  → a)  → β
     end in
     let  take  f  =  perform (M.Prompt f) in
     let  push  f  =  match f () with
     | v → v
     |  effect  (M.Prompt f) k → f  k
     {  take;  push }

  let  push_prompt {push} =  push
  let  take_subcont  {take} =  take
  let  push_subcont k v =
     let  k'  =  Obj.clone_continuation k  in
     continue  k'  v

  let  shift0  p f  =
     take_subcont  p  (fun sk  →
     f  (fun c  → push_prompt p (fun ()  → push_subcont sk  c)))
end
```

Figure 1: Embedding Delimcc in multicore OCaml

straightforward to add and their treatment is standard. Therefore, we elide them. The handler construct in Eff has a finally clause – which is the syntactic sugar and is hence omitted in the Core Eff. Declaring several operations for an effect is certainly natural and convenient. Without the loss of generality one may, however, separate each operation in its own effect, which is what we do for Core Eff: an effect has only one operation, which hence does not have to be named. There is no need in effect declarations either. We do retain the facility to create, at run time, arbitrarily many instances of the effect. In Core Eff, an effect instance alone hence acts as the effect name.

Thus, the Core Eff has integer and arrow types, the type $t_1 \hookrightarrow t_2$ of an effect operation that takes a t_1 value as an argument and produces the result of the type t_2, and the type $t_1 \Rightarrow t_2$ of a handler acting on computations of the type t_1 and producing computations of the type t_2.

The conventional presentation of syntax in Fig. 2 can be also given in a 'machine-readable' form, as an OCaml module signature, Fig.3. The (abstract) OCaml type α repr represents the Core Eff type α of its values. In the same vein, α res represents the type α of Eff computations. The paper [2] likewise

Variables	x,y,z,u,f,r. . .
Constants	c ::= unit, integers, integer operations
Types	t ::= unit | int | t $\rightarrow$ t | t $\hookrightarrow$ t | t $\Rightarrow$ t
Values	v ::= x | c | λx:t. e | op v | h
Handler	h ::= handler v v_v v_o
Expressions (Computations)	e ::= **val** v | **let** x = e **in** e | v v | newp | **with** h handle e

Figure 2: The Core Eff

```
module type Eff = sig
  type α repr                       (* type of values *)
  type α res                        (* type of computation results *)

  type (α,β) eff                    (* effect instance type *)
  type (α,β) effh                   (* effect handler type *)

  (* values *)
  val int: int → int repr
  val add: (int→int→int) repr
  val unit: unit repr

  val abs: (α repr → β res) → (α→β) repr
  val op: (α,β) eff repr → (α →β) repr        (* effect invocation *)
  val handler: (α,β) eff repr →               (* effect instance *)
               (γ→ω) repr →                   (* val handler *)
               (α → (β → ω) → ω) repr →       (* operation handler *)
               (γ,ω) effh repr

  (* computations *)
  val vl: α repr → α res                      (* all values are computations*)
  val let_: α res → (α repr → β res) → β res
  val ($$): (α → β) repr → α repr → β res
  val newp: unit → (α,β) eff res              (* new effect instance *)
  val handle: (γ,ω) effh repr → γ res → ω res
end
```

Figure 3: The syntax and the static semantics of the Core Eff, in the OCaml notation

distinguishes the typing of values and computations, but in the form of two different judgements[11]. A few concessions had to be made to OCaml syntax: We write $(\mathsf{t}_1,\mathsf{t}_2)$ eff for the effect type $\mathsf{t}_1 \hookrightarrow \mathsf{t}_2$ and $(\mathsf{t}_1,\mathsf{t}_2)$ effh for the type $\mathsf{t}_1 \Rightarrow \mathsf{t}_2$ of handlers. We use vl in OCaml rather than val since the latter is a reserved identifier in OCaml. Likewise we spell Eff's let as let_, the Eff application as the infix $$, and give newp a dummy argument. We mark integer literals explicitly: whereas 1:int is an OCaml integer, int 1:int repr is the Core Eff integer literal, which is the Eff value of the Eff type int. We rely on the Higher-Order abstract syntax (HOAS) [8, 15, 4], using OCaml functions to represent Eff functions (hence using OCaml variables to represent Eff variables).

The signature Eff encodes not just the syntax of the Core Eff but also its type system, in the natural-

[11]Since the signature Eff also represents the type system of Eff, in the natural deduction style, one may say that α repr and α res represent a type judgement rather than a mere type.

deduction style. For example, the **val** op and **val** handle declarations in the Eff signature straightforwardly represent the following typing rules from [2, §3], adjusted for Core Eff and the natural deduction presentation:

$$\frac{\vdash_v\ \mathsf{v} : \mathsf{t_1} \hookrightarrow \mathsf{t_2}}{\vdash_v\ \mathsf{op}\ \mathsf{v} : \mathsf{t_1} \to \mathsf{t_2}} \qquad \frac{\vdash_v\ \mathsf{h} : \mathsf{t_1} \Rightarrow \mathsf{t_2} \qquad \vdash_e\ \mathsf{e} : \mathsf{t_1}}{\vdash_e\ \mathsf{with}\ \mathsf{h}\ \mathsf{handle}\ \mathsf{e} : \mathsf{t_2}}$$

The type system has two sorts of judgements, for values $\vdash_v\ \mathsf{v} : \mathsf{t}$ and for computations $\vdash_e\ \mathsf{e} : \mathsf{t}$ – which we distinguish by giving the type t repr to the encoding of Eff values and t res to the encoding of computations. The rules express the intent that effect operation invocations act as functions and that a handler acts as an expression transformer.

The benefit of expressing the syntax and the type system of a language in the form of Eff-like signature – in the so-called *tagles-final style* – and the reason to tolerate concessions to OCaml syntax is the ability to write core Eff code and have it automatically typed-checked (and even getting the types inferred) by the OCaml type checker. For instance, the following module encodes a simple Core Eff example, defining a Reader-like int ↪ int effect that increments its argument by the value passed the environment. The ans expression invokes the operation twice on the integer 1, eventually supplying 10 as the environment.

```
module Ex1(E:Eff) = struct
  open E
  (* A macro to apply to a computation: mere ($$) applies to a value *)
  let ($$$) e x = let_ e (fun z → z $$ x)

  let readerh p =
    handler p (abs (fun v → vl @@ abs (fun s → vl v)))
              (abs (fun x → vl @@ abs (fun k → vl @@ abs (fun s →
                let_ ((add $$ s) $$$ x) (fun z → (k $$ z) $$$ s )))))

  let ans =
    let_ (newp ()) @@ fun p →
    let_ (handle (readerh p) @@
           let_ (op p $$ (int 1)) (fun x →
           let_ (op p $$ x)       (fun y →
           vl y))) (fun hr →
    hr $$ int 10)
end
```

The OCaml type-checker verifies the code is type-correct and infers for ans the type int E.res, meaning ans is a computation returning an int.

There is another significant benefit of the tagless-final style. The signature Eff looks like a specification of a denotational semantics for the language. Indeed, repr and res look like semantic domains – corresponding to the domains V and R from [2, §4], but indexed by types. Then int, abs, op, handle and the other members of the Eff signature are the semantic functions, which tell the meaning of the corresponding Eff value or expression from the meaning of its components. The compositionality is built into the tagless-final approach.

The signature Eff is only the spefication of semantic functions. To define the denotational semantics of the Core Eff we need to give the implementation. It is shown in Fig. 4. The module R implementing Eff is essentially the denotational semantics of Eff given in [2, §4], but written in a different language: OCaml rather than the standard mathematical notation. It is undeniable that the conventional mathematical notation is concise – although the conciseness comes in part from massive overloading and even

```
module REff = struct
  type α repr  =
    | B : α → α repr                          (* OCaml values *)
    | F : (α repr→β res) → (α→β) repr         (* Functions V→R,
                                                     i_arr  in the Eff paper *)
  and _ res   =                                (* Results *)
    | V: ω repr → ω res                        (* Normal termination result *)
    | E: {inst: int; arg:α repr; k:β repr→ω res} → ω res

  let rec lift  : (α repr → β res) → α res → β res = fun f → function
    | V v → f v
    | E ({k;_} as oper) → E {oper with k = fun x → lift f (k x)}

  type (α,β) eff = int
  type (α,β) effh = (unit → α) → β

  (* values *)
  let int (x: int) = B x
  let add : (int→int→int) repr =
    F (function B x → V (F (function B y → V (B (x+y)))))
  let unit = B ()

  let abs f = F f
  let ($$): (α → β) repr → α repr → β res =
    function F f → fun x → f x

  let op: (α,β) eff repr → (α → β) repr = function B p →
    abs (fun v → E {inst= p; arg= v; k = fun x → V x})

  let handler: (α,β) eff repr →               (* effect instance *)
               (γ→ω) repr →                   (* val handler *)
               (α → (β → ω) → ω) repr →       (* operation handler *)
               (γ,ω) effh repr =
     fun (B p) (F valh) (F oph) →
       let rec h = function
         | V v             → valh v
         | E {inst;arg;k} when inst = p →
             let V (F kh) = oph (Obj.magic arg) in
             let (k:β repr → γ res) = Obj.magic k in
             (* Since the handlers are deep, we compose with k with h *)
             kh (abs (fun b → h (k b)))
         (* Relay to an outer handler *)
         | E ({k:_} as oper) → E {oper with k = fun b → h (k b)}
       in abs (fun th → h (th $$ unit))

  let vl v = V v                          (* all values are computations *)
  let let_ : α res → (α repr → β res) → β res = fun e f → lift f e

  let newp: unit → (α,β) eff res =
    let c = ref 0 in
    fun () → incr c; V (B !c)

  let handle: (γ,ω) effh repr → γ res → ω res =
    fun h e → h $$ abs (fun (_: unit repr) → e)
end
```

Figure 4: The denotational semantics of the Core Eff

sloppiness, omitting details like various inclusions and retractions. The OCaml notation is precise. Moreover, the OCaml type-checker will guard against typos and silly mistakes. Since we index the domains by type, there are quite a few simple correctness properties that can be ensured effectively and simply. For example, forgetting to compose the continuation with the handler h in handler leads to a type error.

The denotations of Core Eff are expressed in terms of two semantic domains, of values and results. In [2], the domains are called V and R respectively. We call them α repr and α res, and index by types. The type-indexing lets us avoid many of the explicit inclusions and retractions defined in [2, §4]. In our R implementation, domains are defined concretely, as OCaml values, viz. mutually recursive data types repr and res. Of all the retracts of [2] we only need two non-trivial ones. The first is $\iota_{\rightarrow}$ and $\rho_{\rightarrow}$ in [2], which embeds the functions α repr $\rightarrow$ β res into $(\alpha{\rightarrow}\beta)$ repr. This embedding is notated as F (the inclusion is applying the F constructor and the retraction is pattern-matching on it). The second is the α res $\rightarrow$ β res retract, implemented via the isomorphism with (unit$\rightarrow\alpha$) repr $\rightarrow$ β res, which is then embedded into ((unit$\rightarrow\alpha$)$\rightarrow\beta$) repr. The domain repr does not need the bottom element since values are vacuously terminating, and our denotational semantics is typed, Church-style: we give meaning only to well-formed and well-typed expressions.

We define the domain α res to be nothing bigger than its required retract, the sum expressing the idea that a terminating computation is either a value, V, or an effect operation. The latter is a tuple that collects all data about the operation: the instance, the argument, and the continuation. The lifting of f:α repr $\rightarrow$ α res to the α res domain, written as $f^{\dagger}$ in [2], is notated as lift f in our presentation. The implementation of int, abs, op and the rest of the members of Eff is the straightforward transcription of the definitions from [2]. (We use the higher-order abstract syntax and hence do not need the explicit 'environment' η.) The appearance of Obj.magic comes from the fact that the Core Eff (just like the full Eff) does not carry the effect type in the type of a computation. Therefore, the argument and result types of an effect are existentialized. One may hence view Obj.magic as an implicit retraction into the appropriate α repr domain. The use of Obj.magic is safe, thanks to the property that each effect instance (denoted by an integer) is unique; that is, the instances of differently-typed effects have distinct values.

Having recalled the semantics of Eff, we now turn to the delimited control, and then, in §3.3, to the translation from Core Eff to the Core OCaml with delimited control.

3.2 Denotation of Delimited Control

This section describes the target language of the Eff embdedding, which is OCaml with the delimcc library. As we did with Eff, we reduce the language to the bare minimum, to be called the Core delimcc. The syntax and the static semantics (that is, the type system) is presented in Fig. 5. From now on, we will be using the OCaml rather than the mathematical notation. The style of the presentation should be familiar from §3.1.

The Core delimcc is, in many parts, just like Core Eff, Fig. 3, and is likewise decribed by the OCaml signature. The Core delimcc is a bigger language: besides the unit and int basic types it also has the universal type with the corresponding injection i_univ and projection p_univ; besides ordinary function definitions it has recursive functions absrec. Recursive functions can also be defined in the full Eff; we did not need them for the Core Eff subset. The Core delimcc also has a composite algebraic datatype free, which is a sum whose second summand is a tuple. The data type is represented by the constructors ret and act for the summands, and the deconstructor (eliminator) with_free. The delimcc-specific part [11] is the type of control delimiters, so-called prompts, the operations to create a fresh prompt newpr, set the prompt pushpr and to capture the continuation up to the dynamically closest pushpr, the operation "shift-0" sh0.

```
module type Delimcc = sig
  type α repr
  type α res

  (* values *)
  val int : int → int repr
  val add: (int→int→int) repr
  val unit : unit repr

  type univ                                 (* the universal type *)
  val i_univ : α repr → univ repr
  val p_univ: univ repr → α res

  val abs: (α repr → β res) → (α→β) repr
  val absrec: ((α→β) repr → α repr → β res) → (α→β) repr

  type (α,β) free =
    | Ret of univ repr
    | Act of α repr * (β → (α,β) free) repr
  val ret : univ repr → (α,β) free repr
  val act: α repr → (β → (α,β) free) repr → (α,β) free repr
  val with_free : (α,β) free repr →
                  (univ repr → ω res) →
                  (α repr → (β → (α,β) free) repr → ω res) →
                  ω res

  (* computations *)
  val vl : α repr → α res                    (* all values are computations*)
  val let_ : α res → (α repr → β res) → β res
  val ($$): (α → β) repr → α repr → β res

  (* The delimcc part: prompt and shift *)
  type α prompt
  val newpr: unit → α prompt res
  val pushpr: α prompt repr → α res → α res
  val sh0:    α prompt repr → ((β → α) repr → α res) → β res
end
```

Figure 5: The syntax and the type system of the Core delimcc

The semantics of delimited control is typically presented in the small-step reduction style (see [6, 11]):

```
pushpr p (vl x)                     ⇝ vl x
pushpr p (Cp[sh0 p (fun k → e)])    ⇝ let k = abs (fun x → pushpr p Cp[x]) in e
```

where Cp[x] is the evaluation context with no sub-context pushpr p []. In contrast, we treat the Core delimcc denotationally, giving it semantics inspired by the "bubble-up" approach of [7, 17]. Establishing the formal correspondence with the standard reduction semantics and investigating full abstraction is left for future work.

Our denotational semantics of the Core delimcc, Fig. 6, is (intentionally) quite similar to that for the Core Eff, Fig. 4. It is given in terms of domains α repr of value denotations and α res of expression denotations. The value denotations are the same as in the Core Eff. A terminating expression is either a value V, or a "bubble" E created by sh0. The bubble merely packs the data from the sh0 that created it (the prompt value plus the body of the sh0 operator), along with the continuation k that represents the context of that sh0. All in all, the bubble represents the decomposition of an expression as the sh0 operation embedded into an evaluation context.

The only non-standard parts of the semantics are the denotations of sh0 and pushpr. As was already said, sh0 creates the bubble, by packing its arguments along with the identity continuation representing the empty context. The function lift (essentially let_) – which represents a let-bound expression in the context of the let-body – grows the bubble by adding to it the let-body context. The operation pushpr p "pricks" the bubble (but only if the prompt value p matches the prompt value packed inside the bubble, that is, the prompt value of the sh0 that created the bubble). When the bubble is pricked, the sh0 body hidden inside is releazed and is applied to the continuation accummulated within the bubble – enclosed in pushpr p as behoves to the shift operation. Again, Obj.magic comes from the fact that we do not carry the answer type in the type of a computation. Therefore, the answer type ω is existentialized in the bubble. When the bubble is pricked however, we are sure that the answer-type is actually the type of the pushpr computation. The coercion operation is hence safe.

3.3 Translation from Eff to Delimited Control, and its Correctness

Having formalized the semantics of the Core Eff and the Core OCaml with delimcc, we are now in a position to formally state the translation and argue about its correctness.

The tagless-final style used for the denotational semantics makes it straightforward to express a compositional translation. Indeed, a language is specified as an OCaml signature that collects the declarations of syntactic forms of the language. The interpretation – semantics – is an implementation of the signature. A given signature may have several implementations. For example, the signature Eff (Fig. 3) of the Core Eff had the REff implementation (Fig. 4). Fig. 7 shows another implementation, in terms of the Core delimcc: it maps the types and each of the primitive expression forms of the Core Eff to the types resp. expressions of the Core delimcc. The mapping homomorphically extends to composite Core Eff expressions; such an extension is inherent in the tagless-final approach. The mapping should not depend on any concerete implementation of delimcc: therefore, it is formulated only in terms of the abstract types and methods defined in the Delimcc signature, Fig. 5. In OCaml terms, the translation is represented as a functor, Delimcc $\to$ Eff.

The translation is rather straightforward: α repr and α res domains of Eff map to the corresponding domains of delimcc. An Eff effect instance maps to a delimcc prompt. Most of the Core Eff expression forms (int, add, abs, let_, etc) map to the corresponding Core delimcc forms. Only op and handler of the Core Eff have non-trivial implementation in terms of delimcc: op is just sh0 that creates a bubble with the data about the effect operation. The effect handler interprets those data. Since effect handlers in Eff are deep (that is, after an effect is handled and the expression is resumed, the handler is implicitly re-applied), they correspond to recursive functions in the Core delimcc. Again, the appearance of the universal type in handler comes from the fact that we do not carry the effect type in the type of a computation. In §2.2 we emulated the universal type in terms of reference cells, a well-known folklore[12].

The Translation functor defines, in OCaml notation, the translation of Core Eff types and expressions, which we can notate $\lceil$ t $\rceil$ and $\lceil$ e $\rceil$. The fact the translation deals with typed expressions and the fact the Translation functor is accepted by the OCaml type-checker immediately lead to:

Proposition 1 (Type Preservation) *If* e *is a Core Eff expression of the type* t *(whose free variables,*

[12] http://mlton.org/UniversalType

```
module RDelimcc = struct
  type α repr  =
    | B : α → α repr
    | F : (α repr→β res) → (α→β) repr
  and _ res    =
    | V: α repr → α res
    | E: {prompt: int; body:(γ →ω) repr→ω res; k: γ repr → α res} → α res

  let rec lift  : (α repr → β res) → α res → β res = fun f → function
    | V v → f v
    | E ({k;_} as oper) → E {oper with k = fun c → lift f (k c)}

  (* values *)
  let int (x:int) = B x
  let add : (int→int→int) repr = F (function B x → V (F (function B y → V (B (x+y)))))
  let unit = B ()

  type univ = Obj.t                              (* the universal type *)
  let i_univ : α repr → univ repr = fun x → B (Obj.repr x)
  let p_univ: univ repr → α res  = function B x → V (Obj.obj x)

  let abs f = F f
  let absrec: ((α→β) repr → α repr → β res) → (α→β) repr = fun f →
    abs (fun x → let rec h y = f (abs h) y in h x)

  let vl v = V v                        (* all values are computations *)
  let let_: α res → (α repr → β res) → β res = fun e f → lift f e
  let ($$): (α → β) repr → α repr → β res = function F f → fun x → f x

  type (α,β) free =
    | Ret of univ repr
    | Act of α repr * (β → (α,β) free) repr
  let ret: univ repr → (α,β) free repr = fun x → B (Ret x)
  let act: α repr → (β → (α,β) free) repr → (α,β) free repr = fun v k → B (Act (v,k))
  let with_free : (α,β) free repr → (univ repr → ω res) →
                  (α repr → (β → (α,β) free) repr → ω res) → ω res =
   fun (B free) reth acth → match free with
   | Ret x → reth x
   | Act (a,k) → acth a k

  type α prompt = int
  let newpr: unit → α prompt res =
    let c = ref 0 in
    fun () → incr c; V (B !c)

  let sh0: α prompt repr → ((β → α) repr → α res) → β res =
    fun (B prompt) body → E {prompt;body;k= vl}

  let rec pushpr: α prompt repr → α res → α res = fun (B p) → function
    | V x → V x
    | E{prompt; body;k} when prompt = p →
        let (body:_→_) = Obj.magic body in
        body (abs (fun c → pushpr (B p) (k c)))
         (* Relay to an outer handler *)
    | E ({k;_} as oper) → E {oper with k = fun c → pushpr (B p) (k c)}
end
```

Figure 6: The denotational semantics of the Core delimcc

```
module Translation(D:Delimcc) = struct
  type α repr   = α D.repr
  type α res    = α D.res

  type (α,β) eff  = (α,β) D.free D.prompt
  type (α,β) effh = ((unit → α) → β)

  (* values *)
  let int  = D.int
  let add  = D.add
  let unit = D.unit

  let abs = D.abs

  let op: (α,β) eff repr → (α → β) repr = fun p →
    D.(abs (fun v → sh0 p (fun k → vl @@ act v k)))

  let compose: (β→γ) repr → (α→β) repr → (α→γ) repr = fun fbc fab →
    D.(abs (fun a → let_ (fab $$ a) (fun b → fbc $$ b)))

  let handler: (α,β) eff repr →                (* effect instance *)
               (γ→ω) repr →                    (* val handler *)
               (α → (β → ω) → ω) repr →        (* operation handler *)
               (γ,ω) effh repr =
     fun p valh oph →
       let h = D.(absrec @@ fun h freer →
         with_free freer
           (fun r   → let_ (p_univ r) (fun r → valh $$ r))
             (* Since the handlers are deep, we compose with k with h *)
           (fun v k → let_ (oph $$ v) (fun kh → kh $$ compose h k)))
      in
       D.(abs (fun th →
         let_ (pushpr p ( let_ (th $$ unit) (fun r → vl (ret ( i_univ r)))))
              (fun freer → h $$ freer )))

  let vl   = D.vl
  let let_ = D.let_
  let ($$) = D.($$)

  let newp: unit → (α,β) eff res = D.newpr

  let handle: (γ,ω) effh repr → γ res → ω res =
    fun h e → h $$ abs (fun (_: unit repr) → e)
end
```

Figure 7: Translation from the Core Eff to the Core delimcc

if any, have the types x1:t_1,...), then ⌈ e ⌉ has the type ⌈ t ⌉ (assuming free variables of the types x1:⌈ t_1 ⌉,...).

The proof immediately follows from the typing of the Translation functor.

We thus have two implementations of the Core Eff: the original REff (Fig. 4) and the result of the translation Translation(RDelimcc). Before we can state the main theorem that these two implementations are "the same" and hence the translation is meaning-preserving, we have to verify that the semantic domains of the two denotational semantics are comparable. The REff implementation has α repr and α res domains defined in Fig. 4 whereas the translated one uses α repr and α res from Fig. 6. While the two α repr have the same structure, α res differ slighly. Both are sums, with the identical V component, and the E component being a triple: {inst: int; arg:γ repr; k:β repr→α res} vs. {prompt: int; body:(β →γ) repr→γ res; k: β repr → α res}. Although the first and the third components of the triple are compatible, the middle is not. A moment of thought shows that the only delimcc bubbles possible in the Translation(RDelimcc) implementation are those that come from op, in which case the body of the bubble is **fun** k → vl @@ act v k, or, unfolding the definitions, **fun** k → V (Act (v,k)), with v being the argument arg of the effect operation. Hence the triple {inst;arg;k} can be turned to {prompt= inst;body = (**fun** k → V (Act(arg,k)));k} (and easily retracted back). Thus although α RDelimcc.res domain is 'bigger', to the extent it is used in the Translation(RDelimcc), it is isomorphic to α REff.res. This isomorphism is implicitly used in the main theorem:

Proposition 2 (Meaning Preservation) *A Core Eff value or expression has the same meaning under REff and Translation(RDelimcc) semantics.*

The proof has to verify that types correspond to the same domains in both interpretations and that primitive forms of the Core Eff have the same interpretations. We have already discussed the α repr and α res domains in both semantics. Clearly (α,β) eff type has the same interpretation (integer in both semantics), and so does (α,β) effh. Most of the Core Eff forms have obviously the same interpretation in both semantics. The only non-trivial argument concerns op and handler. The expression op p denotes the function **fun** v → E{inst= p;arg= v;k= **fun** x→V x} under the REff semantics and the function **fun** v → E{prompt= p;body= (**fun** k → V (Act(v,k)));k= **fun** x→V x} under the translation semantics. As we argued earlier, the denotations are the same (keeping our isomorhism in mind).

The handler p valh oph in both interpretations is a function from γ res to ω res. To see that it is the same function, we consider three cases. First, if the argument is of the form V x, both interpretations converge on valh x. If the argument is of the form E {inst;arg;k} (in the REff interpretation) with inst= p, the first interpretation gives oph arg (handler p valh oph ∘ k). In the translation interpretation, the correspodning handled expression has the denotation E {prompt;body;k}, with prompt being equal to p and body being **fun** k → V (Act (arg,k)). Then pushpr p (E {prompt;body;k}) amounts to body (pushpr p ∘ k), which is V (Act (arg,pushpr p ∘ k)). It is then handed over to the recursive function h in Fig. 7, which returns oph arg (h ∘ pushpr p ∘ k). The latter matches the REff denotation. The remaining case is of the handled expression being E {inst;arg;k} (in the REff interpretation) with inst different from the handler's p. The REff intrepretation gives E {inst;arg;handler p valh oph ∘ k}. It is easy to see that the translation interpretation gives the same.

4 Higher-order Effects

The running example from §2.1 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.

5 Evaluation

In this section, we evaluate the performance for Eff 3.1 embedded in OCaml and compare it against the performance of Eff 3.1, compiled with the optimizing backend. For the embedded versions, we consider both the delimcc and the multicore OCaml backends. For the sake of comparison, whenever possible, we also evaluate the performance of equivalent programs written in *pure* OCaml i.e., without the use of effects and handlers.

5.1 N-queens benchmark

The benchmark we consider is the N-queens benchmark. The aim of the benchmark is to place N queens on a board of size N such that no two queens threaten each other. The algorithm involves a backtracking depth-first search for the desired configuration. For this benchmark, we consider the following 6 versions of the N-queens program:

- Exception: A pure version with backtracking implemented using native OCaml exceptions.
- Option: A pure version with backtracking implemented using option type.
- Eff: An impure version of the benchmark compiled using Eff optimizing compiler backend. Backtracking implemented using effect handlers.
- Multicore: An impure version where backtracking is implemented with native effects in multicore OCaml.
- Eff_of_multicore: An impure version of the benchmark implemented in Eff embedded in OCaml using multicore OCaml handlers.
- Eff_of_delimcc: An impure version of the benchmark implemented in Eff embedded in OCaml using delimcc backend.

The code for Exception version is presented in Fig. 8. no_attack returns true if two queens on the board do not threaten each other. available function, given qs, a safe assignment of queens in the first x−1 rows, returns the list of possible safe positions for a queen on the xth row. The function place attempts to safely place n queens, one on each row in a non-threatening configuration on the board of size n. This is done by exploring the possible assignments in a depth-first fashion. If the search along a path is no successful, the control backtracks by raising exception Failure, and the next path is attempted. If successful, the function returns the configuration. The main funtion queens_exception prints a success message if some safe configuration is possible. Otherwise, it prints an error message.

Fig. 9 shows the Multicore version of the n-queens benchmark. We declare an effect 'Select' which is parameterized with a list of elements of some type, which when performed returns an element of that

```
let no_attack (x,y) (x',y') =
  x ≠ x' && y ≠ y' && abs (x − x') ≠ abs (y − y')

let available n x qs =
  let rec loop possible y =
    if y < 1 then possible
    else if List . for_all (no_attack (x, y)) qs then
      loop (y :: possible ) (y − 1)
    else loop possible (y − 1)
  in
  loop [] n

exception Failure

let queens_exception n (* number of queens *) =
  let rec place x qs =
    if x > n then qs else
      let rec loop = function
        | [] → raise Failure
        | y :: ys →
          try place (x + 1) ((x, y) :: qs)
          with Failure → loop ys
      in
      loop ( available n x qs)
  in
  match place 1 [] with
  | res → print_endline "Success!"
  | exception Failure → print_endline "Fail:␣no␣valid␣assignment"
```

Figure 8: Backtracking n-queens benchmark implemented using exceptions.

```
let no_attack (x,y) (x',y') =
  x ≠ x' && y ≠ y' && abs (x − x') ≠ abs (y − y')

let available x qs l =
  List . filter (fun y → List . for_all (no_attack (x,y)) qs) l

effect Select : α list → α

let queens_multicore n =
  try
    let l = ref [] in
    for i = n downto 1 do
      l := i::! l ;
    done;
    let rec place x qs =
      if x = n+1 then Some qs else
        let y = perform @@ Select (available x qs !l) in
        place (x+1) ((x, y) :: qs)
    in place 1 []
  with
  | effect (Select lst) k →
      let rec loop = function
        | [] → None
        | x :: xs →
            match continue (Obj.clone_continuation k) x with
            | None → loop xs
            | Some x → Some x
      in loop lst
```

Figure 9: Backtracking n-queens benchmark implemented using multicore OCaml effect handlers.

type. For placing each queen, in the place function, we perform the effect ‘Select‘ with the list of available positions for the next queen. The effect handler performs backtracking search and explores each of the possibilities by invoking the continuation with different assignments for the position of the next queen. Since continuations in multicore OCaml are one-shot by default, we need to clone the continuation before we resume the continuation. The cost of cloning is linear in the size of the continuation.

5.2 Results

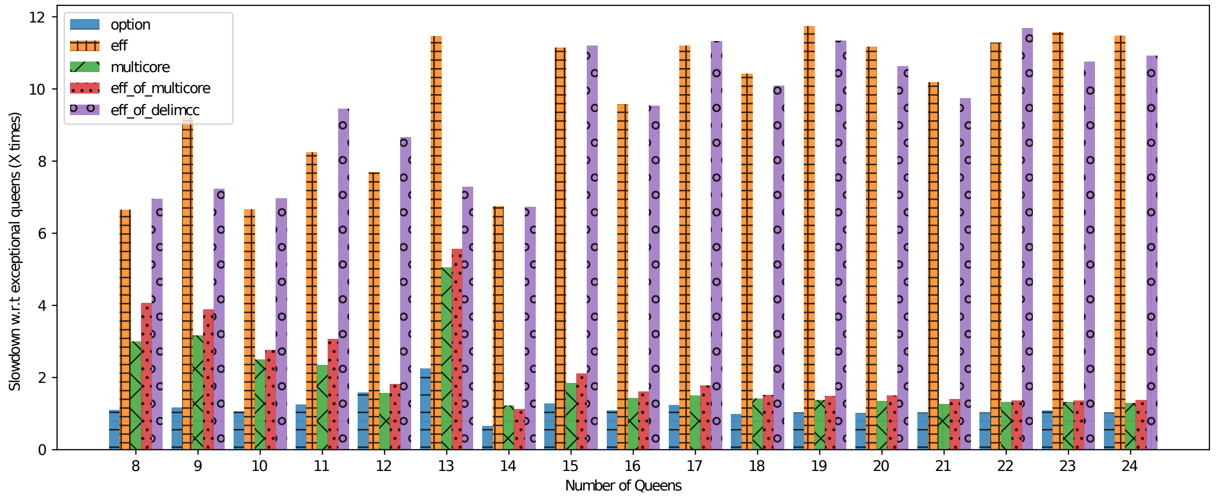

Figure 10: Performance comparison on N-queens benchmark.

Fig. 10 shows the performance of different versions of N-queens benchmark. The results show the running times of each versions normalized to Exception version as we increase the size of the board. The results show that pure OCaml versions perform best and on par with each other. This is unsurprising since these versions do not incur the cost of effect handlers and reifying the continuations. Multicore version performs best among the effectful versions. Multicore OCaml implements effect handlers natively with the help of first-class runtime support for delimited continuations that is fully integrated into the OCaml’s runtime system. As a result, installing effect handlers and continuation capture are cheap operations. We observed that Eff embedding in Multicore OCaml was only $1.3\times$ slower than the exception version. Eff_of_multicore performs marginally slower than Multicore due to boxing overheads.

Eff version and Eff_of_delimcc are comparatively slower than the other versions. This is because delimcc is designed to be an independent library that requires no change to the OCaml compiler and the runtime. The cost of this generality is that delimcc continuation capture and management are more expensive than continuations in multicore OCaml. On average, Eff_of_delimcc version in $8.3\times$ slower than the exception version. However, both the embeddings, Eff_of_delimcc and Eff_of_multicore are faster than native, optimized Eff versions. Eff implementation of handlers is through Free monadic interpretation, incurring the cost of intermediate closures even for pure OCaml code. While the Eff compiler optimizes primitive operations, there are large overheads from the use of Free monad for the rest of the program. On average, the Eff version is $10.2\times$ slower than the exception version.

6 Conclusions and the Further Research Program

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 simple benchmarking indicates that the embedding is generally faster than original Eff programs.

New program for delimited control

Is the various universal types and Obj.magic that appears in our code here and there are inherent or the artifact of an inadequayte interface. Indeed, following the well-established analogy between control operators and exceptions, one may see that push_prompt (also called reset) is essentially equivalent to the following rather particular exception-catching form: **try** expr **with** exc $\to$ exc. Although there are indeed cases for which such limited form of exception-catching is appropriate, most of the type we wish to distinguish the normal and exceptional termination of an expression expr. Likewise we wish to distinguish the normal and shiftful termination of an expression expr in push_prompt p expr, and hence have to work around the restricted interface of push_prompt by defining the sum data type such as free. It is interesting question if a better interface for delimited control can be disigned, without unnecessary restrictions and with simpler typing rules.

New denotational semantic approach to delimited control Establishing the formal correspondence with the standard reduction semantics and investigating full abstraction is left for future work.

Future work: see how our expression of higher-order (dynamic) effects works with effect typing (where the type of an expression tells not only its result but also the effects it may execute.)

Acknowledgments

We are very grateful to Andrej Bauer for introducing us to Eff, for patiently explaining Eff features and design decisions, and for writing some of the sample Eff code in §2.1. We thank Kenichi Asai for helpful discussions.

References

[1] Kenichi Asai & Yukiyoshi Kameyama (2007): *Polymorphic Delimited Continuations*. In: *APLAS*, *Lecture Notes in Ccomputer Science* 4807, pp. 239–254.

[2] Andrej Bauer & Matija Pretnar (2015): *Programming with Algebraic Effects and Handlers*. *Journal of Logical and Algebraic Methods in Programming* 84(1), pp. 108–123, doi:10.1016/j.jlamp.2014.02.001.

[3] Edwin Brady (2013): *Programming and reasoning with algebraic effects and dependent types*. In ICFP [9], pp. 133–144, doi:10.1145/2500365.2500581.

[4] Alonzo Church (1940): *A Formulation of the Simple Theory of Types*. *Journal of Symbolic Logic* 5(2), pp. 56–68.

[5] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop & Anil Madhavapeddy (2015): *Effective concurrency through algebraic effects*. OCaml Users and Developers Workshop.

[6] R. Kent Dybvig, Simon L. Peyton Jones & Amr Sabry (2007): *A Monadic Framework for Delimited Continuations*. *J. Functional Progr.* 17(6), pp. 687–730.

[7] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker & Bruce F. Duba (1986): *Reasoning with Continuations*. In: *Proceedings of the 1st Symposium on Logic in Computer Science*, pp. 131–141.

[8] Gérard Huet & Bernard Lang (1978): *Proving and Applying Program Transformations Expressed with Second-Order Patterns*. Acta Informatica 11, pp. 31–55.

[9] (2013): *ICFP '13: Proceedings of the ACM International Conference on Functional Programming*. ACM Press.

[10] Ohad Kammar, Sam Lindley & Nicolas Oury (2013): *Handlers in action*. In ICFP [9], pp. 145–158, doi:10.1145/2544174.2500590.

[11] Oleg Kiselyov (2012): *Delimited control in OCaml, abstractly and concretely*. Theoretical Computer Science 435, pp. 56–76, doi:10.1016/j.tcs.2012.02.025.

[12] Oleg Kiselyov & Hiromi Ishii (2015): *Freer monads, more extensible effects*. In: Proceedings of the 8th ACM SIGPLAN symposium on Haskell, Vancouver, BC, Canada, September 3-4, 2015, ACM, pp. 94–105, doi:10.1145/2804302.2804319.

[13] Oleg Kiselyov, Chung-chieh Shan & Amr Sabry (2006): *Delimited Dynamic Binding*. In: ICFP, pp. 26–37.

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

[15] Dale Miller & Gopalan Nadathur (1987): *A Logic Programming Approach to Manipulating Formulas and Programs*. In Seif Haridi, editor: IEEE Symposium on Logic Programming, IEEE Computer Society Press, Washington, DC, pp. 379–388.

[16] (2017): *Multicore OCaml: A shared memory parallel extension of OCaml*. Available at `https://github.com/ocamllabs/ocaml-multicore`. Accessed: 2017-03-31 15:17:00.

[17] Michel Parigot (1992): *$\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction*. In: LPAR, Lecture Notes in AI 624, pp. 190–201.

[18] Gordon Plotkin & Matija Pretnar (2009): *Handlers of Algebraic Effects*. In Giuseppe Castagna, editor: Programming Languages and Systems, Lecture Notes in Ccomputer Science 5502, Springer, pp. 80–94, doi:10.1007/978-3-642-00590-9_7.

[19] Gordon D. Plotkin & John Power (2003): *Algebraic Operations and Generic Effects*. Applied Categorical Structures 11(1), pp. 69–94, doi:10.1023/A:1023064908962.