Dynamic effects report

Albert ten Napel

1 Koka

Koka does not have dynamic effects. It does support references but this is seperate from algebraic effects. Koka's references are very similar to the ST monad in Haskell, instead of the t type parameter Koka uses an h which stands for a heap. Koka has a run function which is like the runST function in Haskell, using a higher ranked type the heap effect can be removed. Following are the relevant parts from the Koka standard library.

```
// A reference ‘:ref<h,a>‘ points to a value of type ‘:a‘ in
   heap ‘:h‘.
type ref :: (H,V) -> V

// Allocate a fresh reference with an initial value.
extern inline ref : forall<h,a> (value:a) -> alloc<h> ref<h,a
   > {
  cs inline "new Ref<##1,##2>(#1)";
  js inline "{ value: #1 }"
}

// If a heap effect is unobservable, the heap effect can be
   erased by using the ‘run‘ fun.
// See also: _State in Haskell, by Simon Peyton Jones and
   John Launchbury_.
extern inline run : forall<e,a> ( action : forall<h> () -> <
   st<h> | e> a ) -> e a  {
  cs inline "Primitive.Run<##2>(#1)";
  js inline "((#1)())";
}
```

2 Eff

Operations in Eff are called on instances, these can be made with the new keyword. The new keyword takes as argument the effect to make an instance of:

```
type 'a state = effect
  operation get: unit -> 'a
  operation set: 'a -> unit
end

(* val r : _a state = <instance #6> *)
let r = new state
```

Note that the examples here are written in Eff version 3, the latest version is 4 but it has removed instances because of complexities with the type system. Also note that the types do not mention any effects. Although a type sytem has been formalized[2] it has not been implemented yet.

Now r is an instance of the state effect and the get and set operations can be called on it:

```
(* val f : a -> a = <fun> *)
let f v =
  let x = r#get () in
  r#set v;
  x;;
```

We can then define a handler for the state effect, notice that it takes an instance of the state effect as argument and handles only the get and set operations on that instance:

```
(* val state : a state -> a -> (b => b * a) = <fun> *)
let state r x = handler
  | val y -> (fun s -> (y, s))
  | r#get () k -> (fun s -> k s s)
  | r#set s' k -> (fun _ -> k () s')
  | finally f -> f x
```

We can now handle f with this handler:

```
(* val f_result : int * int = (42, 100) *)
let f_result = with state r 42 handle f 100;;
```

In the previous example the instance was created statically at the top-level. It is inconvenient to have to specify the instances ahead of time so Eff allows one to dynamically create instances:

```
(* val g : a -> int = <fun> *)
let g v =
  if v = 0 then
    let r = new state in
    r#set 10;
    r#get () + 1
  else
    0;;
```

Now we are not able handle the operations because we have no access to the instance from outside of the function!
We can however handle the operations within the function:

```
(* val g' : a -> int = <fun> *)
let g' v =
  if v = 0 then
    let r = new state in
    fst (with state r 42 handle
      r#set 10;
      r#get () + 1)
  else
    0;;

(* in the REPL *)
# g' 0;;
- : int = 11
```

Now the function is pure.
One problem with dynamic instances is that one can propogate the instance outside of the scope where it is created:

```
(* val h : a -> a state = <fun> *)
let h v =
  let r = new state in
  r#set v;
  r;;
```

To fix this problem Eff introduces resources:

```
let ref x =
  new state @ x with
    operation get () @ s -> (s, s)
    operation set s' @ _ -> ((), s')
  end
```

Now the instance carries a state around and an implicit handler is wrapped around the whole program. This means the operations on this instance can never be unhandled but the always have a default handler. You can however override this behaviour with a custom handler.
To get back Koka style effects (without instances) every effect could get a default instance such that any operation called without an instance will be automatically called on the default instance.

3 Instance creation as an effect

In the paper "Eff directly in OCaml"[9] an embedding of Eff in OCaml is defined using delimited continuations. They also describe how to allow dynamic effects without resources by having the creation of instances just be another effect. The "New" effect will take the effect and the default handler as arguments and create a new instance of that effect and wrap the default handler around the program. This results in the same behaviour as Eff without needing resources. Of course the new effect has to be handled so one needs to wrap the use with the handler of the new effect. A practical programming language could wrap this handler of the new effect around every program by default.

In Haskell-like pseudo-code:

```
-- basic state effect
effect State t {
  get : () -> t
  set : t -> ()
}

-- handlers can have a parameter like Koka,
-- in order to make the definition of the state
-- handler easier. The first argument to k
-- is the value of v in the rest of the
-- continuation.
handler state v {
  get () k -> k v v
  set v k -> k v ()
}

-- the effect of creating new instances.
-- the new operation takes an effect e,
-- a handler of e and returns an instance of e.
effect New e {
  new : e -> Handler e -> Inst e
}

-- the handler of the new effect
-- creates a new instance and wraps
-- the default handler around the program
handler handleNew {
  new eff handle k ->
    let inst = newinstance eff in
    handle inst (k inst)
}

-- references use the state handlers as it's default
ref v = new State (state v)

program = handleNew $
  r1 <- ref 10;
  r2 <- ref 100;
  r1.set 1;
  x <- r1.get ();
  r2.set x;
  return (x + 10)
```

4 Further work

- Nobody has formalized a system that with algebraic effects and handlers and instances yet.
- What kind of type-and-effect system would work for this?
- Do the references defined in this way really have the same semantics as ML-style references?
- Are the references defined this way safe, can they be propogated outside their scope?

	Eff[1][2]	Links [3]	Koka[4]	Frank[5]	Idris (effects)[6]	Multicore OCaml [7][8]
Shallow handlers	No	Yes	Yes	Yes	No	No
Deep handlers	Yes	Yes	Yes	With recursion	Yes	Yes
Effect subtyping	Yes	No	No	No	No	No
Row polymorphism	No	Yes	Only for effects	No	No	No
Effect instances	Yes	No	Duplicated labels	No	Using labels	No
Dynamic effects	Yes	No	No	No	No	No
Indexed effects	No	No	No	No	Yes	No
Linear effect handlers	No	No	Yes	No	No	Yes
Linear types	No	Yes	No	No	Uniqueness types	No

References

[1] Bauer, Andrej, and Matija Pretnar. "Programming with algebraic effects and handlers." Journal of Logical and Algebraic Methods in Programming 84.1 (2015): 108-123.

[2] Bauer, Andrej, and Matija Pretnar. "An effect system for algebraic effects and handlers." International Conference on Algebra and Coalgebra in Computer Science. Springer, Berlin, Heidelberg, 2013.

[3] Hillerström, Daniel, and Sam Lindley. "Liberating effects with rows and handlers." Proceedings of the 1st International Workshop on Type-Driven Development. ACM, 2016.

[4] Leijen, Daan. "Type directed compilation of row-typed algebraic effects." POPL. 2017.

[5] Lindley, Sam, Conor McBride, and Craig McLaughlin. "Do Be Do. In: POPL'2017. ACM, New York, pp. 500-514. ISBN 9781450346603, http://dx. doi. org/10.1145/3009837.3009897."

[6] Brady, Edwin. "Programming and Reasoning with Side-Effects in IDRIS." (2014).

[7] Dolan, Stephen, et al. "Effective concurrency through algebraic effects." OCaml Workshop. 2015.

[8] Dolan, Stephen, Leo White, and Anil Madhavapeddy. "Multicore OCaml." OCaml Users and Developers Workshop. 2014.

[9] Kiselyov, Oleg, and K. C. Sivaramakrishnan. "Eff directly in OCaml." ML Workshop. 2016.