Thesis report on dynamic instances

Albert ten Napel

I will list some uses for dynamic instances

1 Local effects

Assume we are given some function f, that takes a numerical function and returns an integer:

```
f : (Int -> Int) -> Int
```

If we call f with a function g, then how can we count how many times g is called? We would like to define the following function:

```
count : ((Int -> Int) -> Int) -> (Int -> Int) -> Int
```

Where count takes such an f and g and returns the amount of times g was called. In basic algebraic effects and handlers:

```
-- define a tick effect to count
effect Tick {
  tick : () -> ()
}

-- a handler that evaluates Tick and returns the final tick count
tickHandler = handler {
  return x -> \s -> s
  tick () k -> \s -> k () (s + 1)
}
```

```
-- implementation of count
count f g =
  (with tickHandler handle
    f (\x -> tick (); g x)) 0

-- will evaluate to 3
count (\g -> g (g (g 10))) (\x -> x + 1)
```

This works for simple functions but will fail when either f or g uses the tick operation. This is because the handler inside of count will handle all tick operation, irregardless of if they are called by count or by f or g:

```
-- will evaluate to 4 instead of 3
count (\g -> tick (); g (g (g 10))) (\x -> x + 1)

-- will evaluate to 6 instead of 3
count (\g -> g (g (g 10))) (\x -> tick (); x + 1)
```

We could attempt to solve this by defining a seperate instance of a Count effect.

```
effect CountTick {
  countTick : () -> ()
}
```

This would only solve the problem if we had some kind of modules such that we could hide the countTick operation. However now we still have the problem of defining Count twice, any previous functions defined for Count will no longer work for CountTick.

To solve this we need to have a notion of local effects. We want to define and handle an instance of the state effect locally such that the operations of this instance cannot be used anywhere else. Dynamic instances allow us to do this:

```
-- define the tick effect
effect Tick {
  tick : () -> ()
}
```

```
-- a handler that, given an instance of Tick,
-- evaluates and returns the final tick count
tickHandler r = handler {
  return x -> \s -> s
  r#tick () k -> \s -> k () (s + 1)
}

-- implementation of count
count f g =
  let inst = new Tick in
  (with tickHandler inst handle
    f (\x -> inst#tick (); g x)) 0
```

The tick handler used inside of count only handles tick operations on the locally generated Tick instance. Nobody else has access to this tick instance and so this solves the problem we had.

1.1 Local exceptions

We can easily model exceptions with algebraic effects if we ignore the continuation in the handler.

```
effect Exception {
  throw : () -> Void
}

exc = new Exception

trycatch f g = with handler {
  exc#throw () k -> g ()
} handle f ()

div a b =
  if b == 0 then
    exc#throw ()
  else
    a / b
```

```
-- return 0 if b == 0
saveDiv a b =
  trycatch
    (\() -> div a b)
    (\() -> 0)
```

Where normally exception handlers can catch all exceptions, with dynamic instances we can have local exceptions that can only be caught when one has access to the instance. The following function will always result in unhandled operations, since the fresh instance inside of notcatchable is not exposed:

```
notcatchable =
  let inst = new Exception in
  inst#throw ()
```

Using dynamic instances we can then have exceptions that can only be caught in a specific way. Catchable returns both a function that might throw and a function that given a function f catches the exception and calls f. Since the instance is not exposed in any way, one can only catch the exception with the function that is returned.

```
catchable =
  let inst = new Exception in
  (
    \x -> if x == 0 then inst#throw () else x,
    \f -> handler { inst#throw () _ -> f () }
  )
```

Generalizing this idea we can have libraries that require the user to call a certain function at some point or else their program won't typecheck.

```
effect Required {
  require : () -> ()
}
```

```
getLibraryFunction =
  let inst = new Required;
  (
    someLibraryFunction,
    onlyHandlerThatHandlesInst
  )
```

When the user calls someLibraryFunction it calls the operation require. Using an effect system this results in an unhandled operation, where the only way to resolve this type error is by wrapping the call to someLibraryFunction with onlyHandlerThatHandlesInst at some point.

Frankly I haven't found any good use for local exceptions yet.

2 Channels

In order show the usefulness of dynamic instances we will take communication channels as a running example. One can either write to or read from a channel. Examples include reading from or writing to a file, socket or standard output, or message passing between processes.

As example programs we take the following two tasks: Task 1. Give a channel c, read a natural number n from c and write 0 to n (excluding) to c. Task 2. Given a channel c and d, read a natural number n from c and write 0 to n (excluding) to d.

2.1 Basic algebraic effects and handlers

In a system with basic algebraic effects and handlers we can view channels as an effect with two operations, read and write. This provides a common interface for all the different kinds of channels.

```
effect Channel {
  read : () -> Nat
  write : Nat -> ()
}
```

Operations have an parameter type and a return type. In the case of read we give the unit value and we get back a natural number from the channel.

In the case of write we give the natural number to write and we get back the unit value.

We can then implement task 1 as following:

```
program1 () =
  n <- read ();
  program1loop 0
    where
      program1loop i =
        if i == n then
          ()
        else
          write i;
          program1loop (i + 1)
```

Here we are first reading from the channel to get the number n, after we are using recursion to loop from 0 to n and writing the current number to the channel each time. Program1 implements the program abstractly, it does not refer to a specific channel and so can be used for any channel that supports the Channel interface. To give an implementation for a specific channel we use handlers:

```
constantHandler x = handler {
  read () k -> k x
  write v k -> k ()
}
```

Here we define a useless channel handler that always returns x on read and ignores writes.

We can run program1 with the constantHandler as follows:

```
program1result = with constantHandler100 handle (program1 ())
```

For a real input/output channel these would be handled at the top-level similar to how the IO monad is handled in Haskell.

If we try to implement task 2 we run in to problems though. In task 2 we are working with two different channels, but we have no way to pass in the channels or to mention which channels we are targetting when calling the operations. We can solve both these problems with instances.

2.2 Instances

Following an implementation of task 2 given that we have instances.

```
program2 c d =
  n <- c#read ();
  program2loop 0
    where
      program2loop i =
        if i == n then
          ()
        else
          d#write i;
          program2loop (i + 1)

constantHandler c x = handler {
  c#read () k -> k x
  c#write v k -> k ()
}

-- given a log function that logs to the standard output
logHandler c = handler {
  c#read () k -> k 0
  c#write v k -> log v; k ()
}

-- calling program 2, using constantHandler for channel c
-- and logHandler for channel d
-- note that we are creating the channels dynamically
program2result =
  let c = new Channel in
  let d = new Channel in
  with constantHandler c 10 handle
  with logHandler d handle
  program2 c d
```

As this example shows, instances allow us to work with multiple channels at once.

2.3 Creating instances