Thesis report on dynamic instances

Albert ten Napel

1 Memoization

Some needed utility functions and the State effect used.

```
empty : NatMap
find : NatMap -> Nat -> Maybe Nat
update : NatMap -> Nat -> Nat -> NatMap

effect State {
  get : () -> NatMap
  set : NatMap -> ()
}

state : IntMap -> (Nat!{State} => Nat)
state s = handler {
  return v -> \s -> v
  get () k -> \s -> k s s
  set v k -> \_ -> k () v
  finally f -> f s
}
```

1.1 Basic algebraic effects

Memoized fibonacci function.

```
fib : Nat -> Nat!{State}
fib n =
  map <- get ();
  case find map n of
    Just m -> m
    Nothing ->
      let result =
        if n < 2 then
          n
        else
          fib (n - 1) + fib (n - 2);
      set (update map n result);
      result

result = with state empty handle
  fib 4 * fib 5
```

Problem: the program cannot use State for other purposes since it's already used by the fibonacci.

1.2 Static instances

We can solve the previously stated problem by using an instance of the State effects specifically used for the fibonacci function.

```
instance State fibState

fib : Nat -> Nat!{fibState}
fib n =
  map <- fibState#get ();
  case find map n of
    Just m -> m
    Nothing ->
      let result =
        if n < 2 then
          n
        else
          fib (n - 1) + fib (n - 2);
      fibState#set (update map n result);
      result

result = with state fibState empty handle
  fib 4 * fib 5
```

Problem: we cannot define a general memoization function since that would require generating an instance for each memoized function.

1.3 Dynamic instances

Dynamic instances solve the previous problem by allowing us to dynamically generate an instance for each memoized function.

```
memo : ((Nat -> Nat) -> Nat -> Nat) -> ((Nat -> Nat), Inst State)
memo f =
  let inst = new State;
  let rec n =
    map <- inst#get ();
    case find map n of
      Just m -> m
      Nothing ->
        let result = f rec n;
        inst#set (update map n result);
        result;
  (rec, inst)

fibR f n =
  if n < 2 then
    n
  else
    f (n - 1) + f (n - 2)
(fib, fibState) = memo fibR

facR f n =
  if n < 3 then
    n
  else
    n * f (n - 1)
(fac, facState) = memo facR

result =
  with state fibState empty handle
  with state facState empty handle
    fib 4 * fac 5
```

Problem: we have to manually manage fibState and facState and they have become global references instead of local.

1.4 Resources

With resources we can give an instance a default handler that manipulates a piece of state. When the operations on the instance remain unhandled they will get handled by this default handler at the top-level. This way we don't have to return the instance from it's scope and so it will remain local, but at the same time we know it won't be unhandled. In fact in our case there is no other way to handle the instance generated in memo, except for it's default handler.

```
memo : ((Nat -> Nat) -> Nat -> Nat) -> (Nat -> Nat)
memo f =
  let inst = new State @ empty {
    get () s -> s @ s
    set s _ -> () @ s
  };
  let rec n =
    map <- inst#get ();
    case find map n of
      Just m -> m
      Nothing ->
        let result = f rec n;
        inst#set (update map n result);
        result;
  rec

fibR f n =
  if n < 2 then
    n
  else
    f (n - 1) + f (n - 2)
fib = memo fibR
facR f n =
  if n < 3 then
    n
  else
    n * f (n - 1)
fac = memo facR
result = fib 4 * fac 5
```