Thesis report on dynamic instances

Albert ten Napel

In order to show how dynamic instances can be used to define local effects. We will follow an example from [4].

Given a function `f` of type $(a \to b) \to b$. `f` takes a function `g` from a to b and returns a value of type b. We would like to count how many times `f g` uses `g`. We will use an effect `Tick` with a single operation `tick` to count. Implementing this without instances would look something like:

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

countf g =
  (with handler {
    return _ -> \n -> n
    tick () k -> \n -> k () (n + 1)
  } handle f (\x -> tick (); g x)) 0
```

Similar to how f_{cnt} is defined in [4], we apply `f` to a function that wraps `g`. This wrapper function, in addition to calling `g`, also calls the tick operation. We then handle the resulting computation by using the familiar function-passing style to implement a stateful computation which increases the state by 1 every time `tick` is called. We start the computation with the initial value 0 and discard the result of calling `f`, we return the resulting state of the computation, which is equal to the amount of times that `g` is called.
This implementation will not work for any `f` and `g` though, since if `f` or `g` call the `tick` operation then these will be handled by the handler in `countf` instead of being passed on.
To solve this problem we will move to a system with dynamic instances.
With dynamic instances we call the operations of an effect on an instance of that effect. Because we are able to create instances dynamically we can

have *local* effects. These effects are created and handled locally and so are unable to interfere with any other effects. We correctly implement `countf` as follows:

```
countf' g =
  inst <- new Tick;
  (with handler {
    return _ -> \n -> n
    inst#tick () k -> \n -> k () (n + 1)
  } handle f (\x -> inst#tick (); g x)) 0
```

Now, because the instance of `Tick` is created locally, it is impossible that `f` or `g` call any operations on this instance and so we are sure that the handler will only handle the operations that are called in the wrapper function around `g`.

Another example where dynamic instances are useful are when modeling external resources such as files. Similar to [5] we will use two effects to model files. First a `Filesystem` effect with operations to open, close and read from files. Secondly a `File` effect that represents a single file with an operation to read from the file.

```
effect Filesystem {
  open : String -> FileHandle
  read : FileHandle -> String
  close : FileHandle -> ()
}

effect File {
  read : () -> String
}
```

To implement the `Filesystem` effect we can create a special instance `fs` which gets implicitly handled at the top-level, where the actual IO effects are performed.

To allow a programmer to safely work with files we can write a wrapper function that opens a file, applies a function to the filehandle and closes it after the function is done, just like how `file` is defined in [5].

```
file path action =
  handle <- fs#open path;
  file <- new File;
  x <- with handler {
    file#read () k ->
      text <- fs#read handle;
      k text
  } handle action file;
  fs#close handle;
  x
```

`file` takes as arguments the path to the file and the action you want to perform using the file. First we open the file using the `open` operation on the `fs` instance which returns a filehandle. Instead of giving the filehandle to the action, we create an instance of the `File` effect. This instance will be an abstract representation of the file. We call `action` on the `File` instance and handle any `read` operations by calling the `read` operation on the `fs` instance using the filehandle. We store the result of the action in `x`, close the filehandle and return `x`.
By using instances in this way we make sure that the programmer does not have access to the actual filehandle, but only to the abstract representation of it in the form of a `File` instance. In the end we can be sure that the file gets closed after the action is done.

TODO: problem of escaping instances through lambda abstractions, solution of resources

References

[1] Levy, PaulBlain, John Power, and Hayo Thielecke. “Modelling environments in call-by-value programming languages.” Information and computation 185.2 (2003): 182-210.

[2] Pretnar, Matija. “An introduction to algebraic effects and handlers. invited tutorial paper.” Electronic Notes in Theoretical Computer Science 319 (2015): 19-35.

[3] 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.

[4] Dariusz Biernacki, Maciej Pirg, Piotr Polesiuk, and Filip Sieczkowski. “Handle with Care: Relational Interpretation of Algebraic Effects and Handlers” POPL 2018

[5] Leijen, Daan. “Algebraic Effect Handlers with Resources and Deep Finalization”