Dealing with the open keyword

July 29, 2016

1 Environments and closures

Our data structure for a program looks like this:

```
type t =
  Int
| Bool
| Var of string
| Let of ...
| Fun of ...
...
```

Upon parsing, we give functions an environment (in order to get lexical scoping), effectively a closure. This means that, during parsing, we keep track of the environment as new definitions are introduced, and stick this environment in a function definition when it is parsed. Then, at run time, again keeping track of an environment, we prefer the environment in the function closure, when evaluating inside one. This works well. A little messing around with tying-the-recursive-knot is needed to deal with mutually recursive definitions, but it works.

We leave the evaluation of module initialisation like `let x = ...` and `let _ = ...` to after the parsing. The whole module is then evaluated, so each RHS will be a value. We avoid out-of-date items in function environments by using a reference, so for example in this code:

```
let a = f ()

let g () = a + 1
```

Here, `f ()` will not be evaluated each time `g` is called, because the closure of `g` uses a reference so that it knows about the post-module-initialisation version of `a`.

This keeps parsing and evaluating separate, which seems best.

2 The problem with 'open'

OCaml has local opens, for example writing `Char.(code 'x' + 1)` or is the same as writing `Char.code 'x' + 1`. These are relatively easy to handle. We introduce a new `LocalOpen of string * t` node in the tree, and upon encountering it, open the module up and bring its names into the environment at top level.

But we cannot really use the same approach for normal Open. Here is an example:

```
open Sys

let x = argv

open Printf

let _ = printf "%s" argv.[0]
```

We would have to represent this as something like

```
Open("Sys",
     Struct [‘let x = argv‘; Open("Printf",
                                  Struct [‘let _ = printf "%s" argv.[0]‘])])
```

This parenthesised structure `Open(x, ...Open(y, ....))` is awkward to work with, and involves the creation of fake `Struct` items, all of which is out of kilter with the OCaml parse tree (which treats `open` as a structure item).

Here are some options:

1. Introduce a system for looking through a struct in the evaluator, and instead of evaluating the next item which is not a value, go through and do any opens, then evaluate the next item which is not a value, keeping proper track of the env.
2. Try to do the opens at parse time. But this seems to rely on doing module initialisation at parse time, which is horrid, making the parser depend on the evaluator. As an alternative, we could rewrite all the vars in the code (e.g from argv to Sys.argv), and save the module intiialisation until later. We could arrange later for the rewriting not to be displayed to the user by altering the definition of `Var`. The 'opens' then become no-ops to the evaluator.

Current best solution:

- Remove the abortive attempt at module initialisation within the evaluator. Too many downsides
- Consider scheme 1 above in detail. Scheme 2 might be for later, as an efficiency improvement.