Chapter 14

Encoding recursion

14.1 Fixpoint combinators

We have seen that we do not have recursion in ASL. However, it is possible to encode recursion by defining a *fixpoint combinator*. A fixpoint combinator is a function F such that:

$$F\ M \text{ is equivalent to } M\ (F\ M) \text{ modulo the evaluation rules.}$$

for any expression M. A consequence of the equivalence given above is that fixpoint combinators can encode recursion. Let us note $M \equiv N$ if expressions M and N are equivalent modulo the evaluation rules. Then, consider `ffact` to be the functional obtained from the body of the factorial function by abstracting (i.e. using as a parameter) the `fact` identifier, and `fix` an arbitrary fixpoint combinator. We have:

`ffact` is `\fact.(\n. if = n 0 then 1 else * n (fact (- n 1)) fi)`

Now, let us consider the expression $E =$ (`fix ffact`) `3`. Using our intuition about the evaluation rules, and the definition of a fixpoint combinator, we obtain:
$E \equiv$ `ffact (fix ffact) 3`
Replacing `ffact` by its definition, we obtain:
$E \equiv$ `(\fact.(\n. if = n 0 then 1 else * n (fact (- n 1)) fi)) (fix ffact) 3`
We can now pass the two arguments to the first abstraction, instantiating `fact` and `n` respectively to `fix ffact` and 3:
$E \equiv$ `if = 3 0 then 1 else * 3 (fix ffact (- 3 1)) fi`
We can now reduce the conditional into its `else` branch:
$E \equiv$ `* 3 (fix ffact (- 3 1))`
Continuing this way, we eventually compute:
$E \equiv$ `* 3 (* 2 (* 1 1))` $\equiv$ `6`

This is the expected behavior of the factorial function. Given an appropriate fixpoint combinator `fix`, we could define the factorial function as `fix ffact`, where `ffact` is the expression above.

Unfortunately, when using call-by-value, any application of a fixpoint combinator F such that:

$$F\ M \text{ evaluates to } M\ (F\ M)$$

leads to non-termination of the evaluation (because evaluation of $(F\ M)$ leads to evaluating $(M\ (F\ M))$, and thus $(F\ M)$ again).

We will use the Z fixpoint combinator defined by:

$$Z = \lambda f.((\lambda x.\ f\ (\lambda y.\ (x\ x)\ y))(\lambda x.\ f\ (\lambda y.\ (x\ x)\ y)))$$

The fixpoint combinator Z has the particularity of being usable under call-by-value evaluation regime (in order to check that fact, it is necessary to know the evaluation rules of λ-calculus). Since the name `z` looks more like an ordinary parameter name, we will call `fix` the ASL expression corresponding to the Z fixpoint combinator.

```
#semantics (parse_top
#        "let fix be \\f.((\\x.f(\\y.(x x) y))(\\x.f(\\y.(x x) y)));");;
ASL Value of fix is <fun>
- : unit = ()
```

We are now able to define the ASL factorial function:

```
#semantics (parse_top
#        "let fact be fix (\\f.(\\n. if = n 0 then 1
#                                   else * n (f (- n 1)) fi));");;
ASL Value of fact is <fun>
- : unit = ()

#semantics (parse_top "fact 8;");;
ASL Value of it is 40320
- : unit = ()
```

and the ASL Fibonacci function:

```
#semantics (parse_top
#        "let fib be fix (\\f.(\\n. if = n 1 then 1
#                                  else if = n 2 then 1
#                                       else + (f (- n 1)) (f (- n 2)) fi fi));");;
ASL Value of fib is <fun>
- : unit = ()

#semantics (parse_top "fib 9;");;
ASL Value of it is 34
- : unit = ()
```

14.2 Recursion as a primitive construct

Of course, in a more realistic prototype, we would extend the concrete and abstract syntaxes of ASL in order to support recursion as a primitive construct. We do not do it here because we want to keep ASL simple. This is an interesting non trivial exercise!