SUPPLEMENTAL MATERIAL — A PRIMER ON TAGLESS-FINAL PROGRAMMING IN HASKELL

Tagless final programming in Haskell is conventionally done using type classes, A *type class* is used to create *generic programs* which assume that only a minimum set of operations is defined for an unknown type `t`. In the following, we show a minimal working example of type-class which defines a DSL composed of two operations, `add` and `isZero`, for an unknown type `t`; we will call this class of types `Number`:

```haskell
class Number t where
  add :: t -> t -> t                  -- takes two numbers and returns a number
  isZero :: t -> Bool                 -- takes one number and returns a Boolean value
```

Consequently, one could define generic functions that accept as arguments of types that belong to the class `Number`:

```haskell
multiplyByTwo :: (Number t) => t -> t    -- function signature
multiplyByTwo x =                        -- function body
  if isZero x                            -- checks if x is Zero
    then x
    else add x x
```

The notation ’`(Number t) ⇒`’ can be interpreted as a contract for the `multiplyByTwo` function, which can thus be used only if arguments’ type `t` belongs to the `Number` type-class. In fact, `multiplyByTwo` is just a generic program which will be interpreted depending on the implementation of `add` and `isZero`, which is chosen by the compiler depending on the type inferred. To make a particular type belong to a specific type-class, and thus participate in type-inference process, the `instance` directive must be used; for example, here we make the standard `Int` type a member of the `Number` type-class:

```haskell
instance Number Int where
  add x y = x + y
  isZero 0 = True                     -- isZero is defined by pattern matching
  isZero _ = False
```

Tagless final embedding is done by creating a type-class which is parametric with respect to the type of semantic intepreter; this class is conventionally called `Symantics` because it allows to create a concrete syntax for the embedding and provides a way to type the semantics combinators that will be retargeted to each domain. Here we define a a simply typed lambda calculus[1] for integer arithmetic [1]; the type-class parameter `repr` is a type function which will be reified when interpreting the expression:

```haskell
  class Symantics repr where
      int :: Int -> repr Int
      add :: repr Int -> repr Int -> repr Int
      lam :: (repr Int -> repr Int) -> repr (Int -> Int)
      app :: repr (Int -> Int) -> repr Int -> repr Int
```

[1] Where `lam` is the *lambda* abstraction, `app` is function application, while `int` and `add` are two combinators.

A syntax that uses combinators such as `lam` and `app` is called a *higher-order abstract syntax* (HOAS) since it can model not only a representation of a simple type (such as `Int`) but also of a function of that type (e.g., `Int → Int`)[2].

To define multiple interpretations of the above language, one introduces different concrete definitions of the type variable `repr`. For example, here we define a concrete definition, a type named `R`, that allows to interpret the expression `v0` and `r0` following conventional arithmetic (a sort of recursive "identity" interpreter);

```haskell
newtype R a = R {valueOf :: Integer}

instance Symantics R where
    int x     = R x
    add e1 e2 = R (valueOf e1 + valueOf e2)
    lam f     = R (valueOf . f . R
    app e1 e2 = R ((valueOf e1) (valueOf e2))

eval = valueOf  -- the "identity" interpreter

v0 = add (int 1) (add (int 2) (int 2)) -- type is repr Int
f0 = lam (\x → add  x (int 22))       -- type is repr Int → repr Int
r0 = eval (app f0 v0)                  -- type is Int, value is 27
```

The semantic interpretation of `v0` and `r0` can be augmented with something completely different. For example, we could instantiate `repr` such that some set of abstract interpretation is done on the same expressions; here we create an instance of `repr` that checks whether the resulting integer is even or odd:

```haskell
data Parity
    = Even
    | Odd
    deriving (Show)

data PRepr a
    = P Parity
    | F (PRepr Int → PRepr Int)

instance Symantics PRepr where
    int x =
        if (mod x 2 == 0)
            then P Even
            else P Odd
    add (P Even) (P Even) = P Even
    add (P Odd) (P Odd) = P Even
    add _ _ = P Odd
    lam f = F f
    app (F f) e0 = f e0

P r1 = (app f0 v0) :: (PRepr Int)
```

[2]In this work, we are not going as far as declaring an higher-order syntax with `lam` and `app` combinators because standard Haskell syntax will do just fine for our purposes. Such feature, however, becomes essential when one wants to use higher order functions such as `map` and `fold` to define a circuit.

In the above code, `r1` final value is `Odd` which corresponds to the parity of 27. Note that `v0` and `f0` are the same as the example before, but the meaning of the result has changed given the new `Symantics` instance.

REFERENCES

[1] Oleg Kiselyov. 2012. *Typed Tagless Final Interpreters.* Springer Berlin Heidelberg, Berlin, Heidelberg, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3