Haskino Recursion Notes

Mark Grebe

April 16, 2017

1 Overview

These notes document applying the Step/Done/IterLoop transformations to the Haskino DSL and transformation plugin. We show the recursion transformations in the shallow DSL, applying the pass doing the recursion pass before we handle the shallow to deep transformations later in the process.

2 First Example

Starting with a typical iteration example on the Arduino, we will blink an LED a specified number of times in Haskino.

```
led = 13
button1 = 2
button2 = 3

blink :: Word8 -> Arduino ()
blink 0 = return ()
blink t = do
    digitalWrite led True
    delayMillis 1000
    digitalWrite led False
    delayMillis 1000
    blink $ t-1
```

Using the methodology David developed, we start the transformation of the recursive bind to an imperative while loop. We need to modify the data structures and functions used for transformation slightly due to the monadic nature of the Haskino DSL. The base types are the following:

```
data Iter a b
    = Step a
    | Done b

step :: a -> Iter a b
step s = Step s

done :: b -> Iter a b
done d = Done d

iterLoop :: (a -> Arduino (Iter a b)) -> Arduino a -> Arduino b
```

We start by adding a wrapper function to insert our `IterLoop` function:

```
blink :: Word8 -> Arduino ()
blink = iterLoop blink2

blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI 0 = done <$> return ()
blinkI t = done <$> do
    digitalWrite led True
    delayMillis 1000
    digitalWrite led False
    delayMillis 1000
    blink $ return $ t-1
```

Now we can translate the `done` and recursive call to a `step` using the following rule:

```
forall f x.
    done <$> (g >>= (f x))
        =
    step <$> (g >>= (return x))
```

With our example, it then becomes:

```
blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI 0 = done <$> return ()
blinkI t = step <$> do
    digitalWrite led True
    delayMillis 1000
    digitalWrite led False
    delayMillis 1000
    return $ t-1
```

Now we can translate to the shallow Haskino DSL's imperative while command:

```
blink2' t = do
    t' <- while t (\ x -> not(x == 0)) (\ x -> do
        if (x==0)
          then
            digitalWrite led True
            delayMillis 1000
            digitalWrite led False
            delayMillis 1000
            return $ x-1
          else
            return 0
        )
    if (t' == 0)
      then return ()
      else return ()
```

As this example only has one `done` instance and one `step` instance, the `if` structures in the body of the `while` and at the end of the function are redundant. Had there been multiple `done` or `step` instances, additional conditional tests would be required in the body of the loop to determine which instance's code would be executed. The transformation could determine if there are single instances and optimize the redundant `if` structures out.

3 Second Example

Our second example deals with another form of iteration that is typical in systems that deal with hardware response. In this example we want a function that waits for a button to be pressed. This is detected by the return from a Haskino read of a digital pin becoming True. The function would be written in a shallow, tail recursive style as follows:

```
wait :: Arduino ()
wait = do
  b <- digitalRead button1
  if b then return () else wait
```

Functions of these type require us to use another wrapper function, so that a loop binding is created that may be tracked and used to determine when the iteration terminates. This is unique to our monadic example, since the termination condition is being determined based on an effect that is input during the iteration.

We add a boolean parameter to the work function, and call it with a wrapper function which uses an initial value of `True` for the parameter.

```
wait :: Arduino ()
wait = wait' True

wait' :: Bool -> Arduino ()
wait' i = do
     b <- digitalRead button1
     if b then return () else wait' True
```

As we did with the first example, we can add a wrapper function and our iterative data structure:

```
wait :: Arduino ()
wait = wait' True

wait' :: Bool -> Arduino ()
wait' = iterLoop $ waitI

waitI :: Bool -> Arduino (Iter Bool ())
waitI i = done <$> do
     b <- digitalRead button1
     if b then return () else wait' True
```

An additional rule allows us to move the done through the monadic binds.

```
forall (f :: Arduino a) (k :: a -> Arduino b).
     done <$> (f >>= k)
            =
     f >>= (done <$> k)
```

Applying the rule transforms the wrapped function to:

```
waitI :: Bool -> Arduino (Iter Bool ())
waitI i =  do
     b <- digitalRead button1
     done <$> if b then return () else wait2' True
```

We can then use the following rule to move the `done` into the `then` and `else` branches of the conditional:

```
forall (c :: Bool) (t :: Arduino a) (f :: Arduino a).
     done <$> (if c then t else f)
           =
     if c then done <$> t else done <$> f
```

Which leaves the example as follows:

```
waitI :: Bool -> Arduino (Iter Bool ())
waitI i = do
     b <- digitalRead button1
     if b then done <$> return () else done <$> wait2' True
```

Once again we can translate the `done` and recursive call to a `step` to give us:

```
waitI :: Bool -> Arduino (Iter Bool ())
waitI i = do
     b <- digitalRead button1
     if b then done <$> return () else step <$> return True
```

Now we can translate to the shallow Haskino DSL's imperative while command:

```
wait'' t = do
    b' < - while True (\ x -> x) (\ x -> do
        b <- digitalRead button1
        if b then return False else return True)
    if b'
      then return ()
      else return ()
```