ShallowDeepPlugin Documentation

Mark Grebe

February 28, 2017

1 Overview

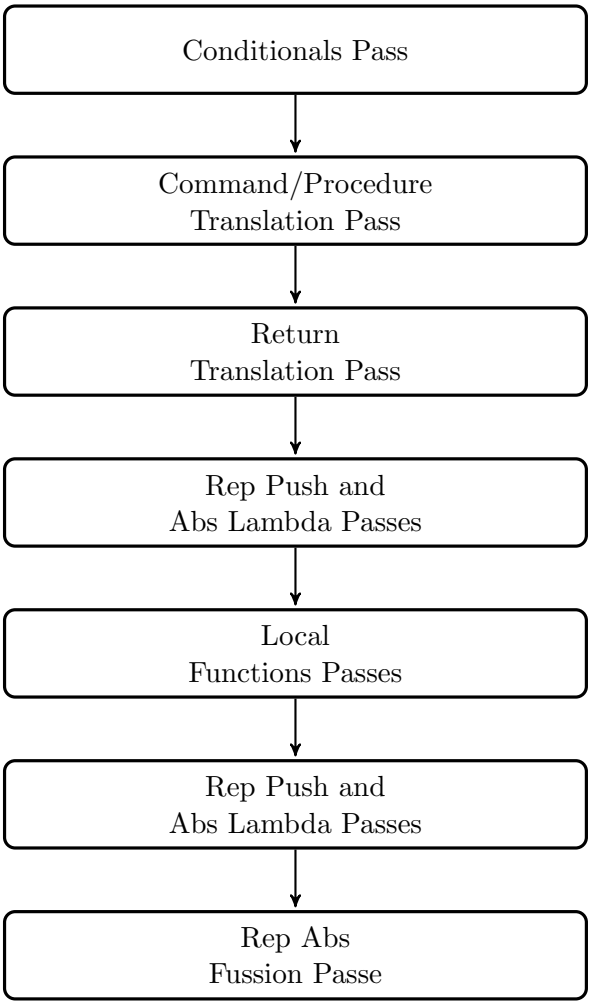
The Shallow Deep Plugin is used to transform embedded DSL's using a Remote Monad form from a shallow embedding to a deep embedding. It currently operates on a per module basis, transforming all DSL functions present in the module.

The plugin is located in System/Hardware/Haskino/ShallowDeepPlugin[.hs]. It is activated in compiling a Haskell module by using the flags -fplugin=System.Hardware.Haskino.ShallowDeepPlugin and -fenable-rewrite-rules.

There are several example test files to use for testing the plugin in System/Hardware/Haskino/SamplePrograms/Rewrite. There are both shallow and deep versions of the same program in separate files (i.e. TwoButton.hs and TwoButtonE.hs). The main function in each of the shallow modules may be run to test the plugin. It reifies, using the Haskino show function, both the deep version and the transformed shallow version, and then compares the resulting strings.

2 Pass Structure

The following diagram shows the order of the passes used in the transformation plugin.



2.1 Conditionals Pass

The conditionals pass transforms Haskell if-then-else expressions into the DSL's embedded ifThenElseE construct. It looks for two armed Case expressions with arms of False and True, which return a type of Arduino a: It performs the equivalent of the following rule (which is not possible with Haskell rules, as the left side is not a function application):

```
forall (b :: Bool) (t :: Arduino a) (e :: Arduino a)
  if b then t else e
    =
  abs_ <$> ifThenElseE (rep_ b) (rep_ <$> t) (rep_ <$> e)
```

For the fmap application of rep to both the then and else expressions, it also does the

following transformation:

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

This ensures that the rep is TBD. The Conditionals Pass is implemented in ShallowDeepPlugin/CondPass.hs.

2.2 Command/Procedure Translation Pass

The Command/Procedure Translation pass transforms all DSL command and procedures from their shallow form to their deep forms.

An example of a command transformation rule is shown below:

```
forall (p :: Word8) (b :: Bool).
  digitalWrite p b
    =
  digitalWriteE (rep_ p) (rep_ b)
```

An example of a procedure transformation rule is shown below:

```
forall (p :: Word8).
  digitalRead p
    =
  abs_ <$> digitalReadE (rep_ p)
```

This pass was originally performed by GHC rules as shown above. However, this pass has now been automated, and only requires a table of shallow/deep command and procedure pairs, and all instances of the shallow DSL elements in the table are transformed to the deep element. An example of the table follows:

```
xlatList = [ XlatEntry (thNameToId 'System.Hardware.Haskino.digitalRead)
                       (thNameToId 'System.Hardware.Haskino.digitalReadE)
           , XlatEntry (thNameToId 'System.Hardware.Haskino.digitalWrite)
                       (thNameToId 'System.Hardware.Haskino.digitalWriteE)
           ]
```

The Commands/Procedures Transformation Pass is implemented in ShallowDeepPlugin/CommProcPass.hs.

2.3 Return Function Translation Pass

The Return Function Translation pass transforms DSL monadic returns through the equivalent of the following rule:

```
forall (x :: a => ExprB a)
  return a
    =
  abs_ <$> return (rep_ x)
```

This transforms returns in the same way that DSL commands and procedures were transformed by the Command/Procedure Transformation Pass. The Returns Transformation Pass is implemented in ShallowDeepPlugin/ReturnsPass.hs.

2.4 Rep Push and Abs Lambda Passes

The Rep Push and Abs Lambda passes are used to manipulate the worker wrapper functions which were inserted by the previous passes, transforming shallow expression functions to deep, and moving the functions for possible fusion in the last pass.

The Rep Push pass is currently performed by rules, and it phase it occurs in is controlled by marking the rules with a [1]. An example of a rep push rule is shown below. One of these rules needs to be defined for each of the DSL's Expr operations:

```
forall (b1 :: Bool) (b2 :: Bool).
  rep_ (b1 || b2)
    =
  (rep_ b1) ||* (rep_ b2)
```

The Rep Push pass is currently executed in the plugin by running the rules1Pass. An attempt to write this pass without rules has been made also, and is located at ShallowDeepPlugin/RepPushPass.hs. Like the CommProcPass, it contains a table of the Expr operations for transformation. (In the example rule about this pair consists of the Template Haskell names for the `(||)` and `(||*)` operators. However, this prototype pass does not work for all of the DSL operators, specifically those using Type Families, such as `(>*)` and `(==*)`. A solution will require generating Cast's and Coercion's, which is yet to be investigated.

```
forall (f :: Arduino a) (g :: a -> Arduino (Expr b)) (k :: b -> Arduino c).
    (f >>= (abs_ <$> g)) >>= k
           =
    (f >>= g) >>= k . abs_
```

```
forall (f :: Arduino a).
    (\x -> F[x]).abs
       =
    (\x' -> let x=abs(x') in F[x])
```

2.5 Local Functions (Binds) Passes

2.6 Rep Abs Fusion Pass

This final pass fuses the rep and abs pairs that have been moved next to each other by the other passes. This is currently performed by two rules and the phase they occur in are controlled by marking the rules with a [0]. Two rules are required, one for direct application, and one for fmap application.

```
forall x.
  rep_(abs_(x))
    =
  x

forall (m :: Arduino (Expr Bool)).
  rep_ <$> (abs_ <$> m)
    =
  m
```

A non-rules based version of this pass has also been created, and is located in ShallowDeepPlugin/RepAbsFusePass.hs

3 Open Issues

- Rules must be present in the module being compiled - This appears to be GHC bug. I have searched for rules defined in imported modules in all of the three locations used by Hermit, and the rules are not present in any of them. Need to do a simplified version and report a bug.
- Complete RepPushPass.hs to handle Expr operations defined with Type Families. This requires understanding of how to generate Cast's and Coersion's in Core.
- Determine how to kick off transformation. Should current module strategy be used, or should all functions called from send(), compile() or show() be transformed?.