ShallowDeepPlugin Documentation

Mark Grebe

March 1, 2017

1 Overview

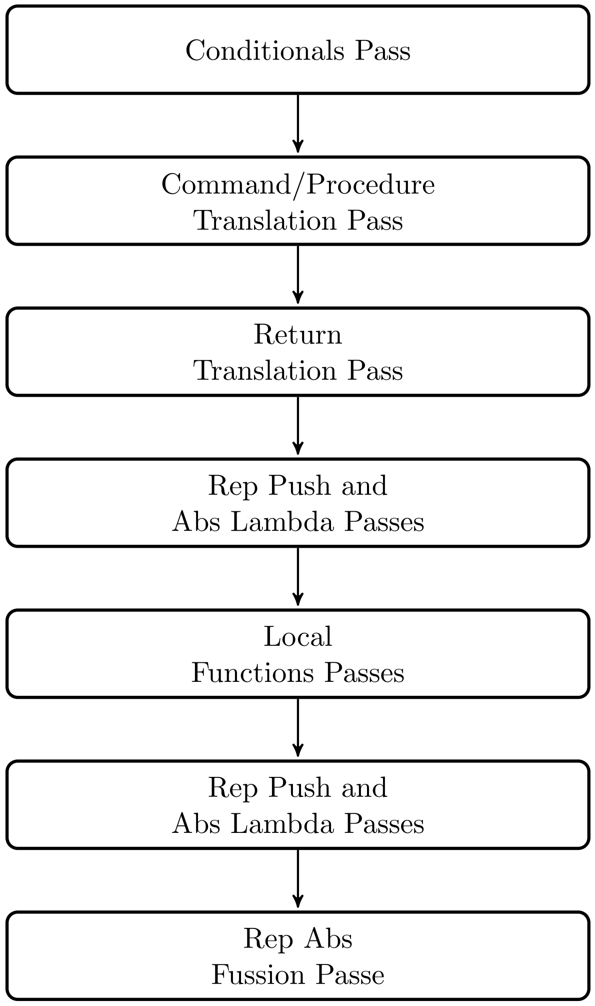
The Shallow Deep Plugin is used to transform embedded DSL's using a Remote Monad form from a shallow embedding to a deep embedding. It currently operates on a per module basis, transforming all DSL functions present in the module.

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. Each of the passes is described in more detail in the remaining subsections of this section. In addition, a dumpPass is defined for debugging purposes, allowing the core to be dumped before and after a pass.


Conditionals Pass
Command/Procedure Translation Pass
Return Translation Pass
Rep Push and Abs Lambda Passes
Local Functions Passes
Rep Push and Abs Lambda Passes
Rep Abs Fussion Passe

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 bodies of 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 is moved to the final monadic expression in the bind chain, which will fuse with the abs that will be placed there by subsequent transformation phases. 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 the 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.

The Abs Lambda pass performs the equivelent of two rules. The first of these uses a varient of the 3rd monad law to move the abs function through a bind to be composed with the continuation.

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

The second rule then takes the composed abs and pushes it into a lambda, as described by the following rule:

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

The plugin pass does this by changing the type of the lambda argument from `a` to `Expr a`, and then replacing all occurances of `x` in the body of the lambda with `abs(x')`.

2.5 Local Functions (Binds) Passes

There are currently three passes used to transform local functions. These passes operate on functions local defined in the module with a return type of Arudino a.

The first of these, BindRetPass.hs changes the return type of the these functions in the function signature, and then fmaps a `rep` to the body to change it's type, as was done with the then and else branches of conditionals.

The second phase, BindArgPass.hs changes the type of each of the arguments of the functions (i.e. `x`) from `a` to `Expr a` in the function signature, and then replaces any occurences of `rep x` with `x`.

The final phase, BindAppPass.hs, changes the types at the application sites of each of the local functions. They are transform similarly to how command, procedures and returns are transformed, with the following rule representing how a function with two arguments are returned.

```
forall (x1 :: a) (x2 :: b) (f :: a -> b -> Arduino c)
  f x1 x2
    =
  abs_ <$> (f (rep_ x1) (rep_ x2)
```

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?.
- Should Haskino DSL be changed to unify the dicotomy of Arduino (Expr a) and Arduino () values by by adding an Expr () to the expression language?

4 Reason for Ordering of Passes

The reasoning as to why there are two instances of the RepPush/AbsLambda passes, and the the local function pass is done after the others can be shown with a small example.

Consider the following local function example:

```
myRead1 :: Word8 -> Arduino Bool
myRead1 p = do
    delayMillis 100
    a <- digitalRead (p+1)
    return (not a)
```

After the command/procedure translation, return translation, and the first repPus/AbsLambda pass the function will look like:

```
myRead1 :: Word8 -> Arduino Bool
myRead1 p = do
```

```
    delayMillisE rep_100
    a <- digitalReadE ((rep_ p) + (rep_ 1))
    return (notB a)
```

It can be seen that these transformations have created the function argument `(rep_ p)` to the `digitalReadE` function. This prepares the function for the BindArgPass. When we change the type of the `p` parameter, we get:

```
myRead1 :: Expr Word8 -> Arduino Bool
myRead1 p = do
    delayMillisE rep_100
    a <- digitalReadE (p + (rep_ 1))
    return (notB a)
```

Performing the BindArgPass first would not work as designed, as there would not be an instance of `(rep_ p)`. The pass could be written differently however, and if done first, changing occurrences of `p` to `(abs_ p)`.