Rewriting a Shallow DSL using a GHC Compiler Extension

Mark Grebe
David Young
Andy Gill
Information and Telecommunication Technology Center
The University of Kansas, USA
first.last@ittc.ku.edu

Abstract

Embedded Domain Specific Languages are a powerful tool for developing customized languages to fit specific problem domains. Shallow EDSLs allow a programmer to program using many of the features of a host language and its syntax, but sacrifice performance. Deep EDSLs provide better performance and flexibility, through the ability to manipulate the abstract syntax tree of the DSL program, but sacrifice syntactical similarity to the host language. Using Haskino, an EDSL designed for small embedded systems based on the Arduino line of microcontrollers, and a compiler plugin for the Haskell GHC compiler, we show a method for combining the best aspects of shallow and deep EDSLs. The programmer is able to write in the shallow EDSL, and have it automatically transformed into the deep EDSL. This allows the EDSL user to benefit from powerful aspects of the host language, Haskell, while meeting the demanding resource constraints of the small embedded processing environment.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Translator writing systems and compiler generators*; Source code generation;

Keywords Haskell, EDSL, GHC, Transformations, Arduino

ACM Reference Format:
Mark Grebe, David Young, and Andy Gill. 2017. Rewriting a Shallow DSL using a GHC Compiler Extension. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3136040.3136048

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136048

1 Introduction

Small, resource constrained embedded systems provide a challenge to programming with high level functional languages. Their small RAM and permanent storage resources make it impossible to run languages like Haskell directly on them. An alternative to using a high level language directly on such systems, is to use an Embedded Domain Specific Language (EDSL). Using an EDSL allows the user to write code using a subset of the look, feel, and semantics of the host language.

Embedded domain specific languages come in two flavors, shallowly embedded and deeply embedded. Shallowly embedded DSLs compute values directly, while deeply embedded DSLs build an abstract syntax tree of computations. With shallow EDSLs, values are computed directly, and chaining together computations requires the involvement of the host language. The syntax and semantics of a shallow EDSL are much closer to the syntax and semantics of a host language. The result of a computation in a deep EDSL is a structure, which may be used to cross-compile the computation before being evaluated. This ability does come at a cost, as deep EDSL's often require special syntactic notations for language features such as control structures.

Table 1. EDSL Options

	Native Execution/ Interpretation	**Code Generation/ Compilation**
Shallow EDSL	hArduino Blank Canvas Haxl • Ease of development • Quick turnaround	?????
Deep EDSL	 • Debugging	Kansas Lava Feldspar Ivory • Performance • Resource Optimization

Table 1 illustrates a subset of the EDSLs hosted by Haskell, our host language. The vertical axes is divided into shallowly and deeply embedded EDSL's. The horizontal axis describes the method of executing the computation on the target system. EDSL's in the first column either execute

the computation natively in the host language, or package the computation for execution in another form, such as a bytecode, for interpretation either locally or remotely. The systems in the second column use the host representation of the computation to generate code in another language, and compile that language for execution on the target system. One way of packaging the computation for remote interpretation, as is done by systems in the first column, and which is used in much of our research, is to use the Remote Monad [15] design pattern. This design pattern allows for bundling of computations to be sent to a remote target.

Shown in the upper left quadrant of the table are EDSLs which share the attributes of being shallowly embedded DSL's, implemented using the Remote Monad design pattern. hArduino [9] is an EDSL used to program small embedded systems based on the Arduino series of boards. It is shallowly embedded, and may only be used while tethered to a host with a USB cable. It is the predecessor to our Haskino system. Blank Canvas [14] is a shallowly embedded EDSL which allows users to program interactive images in a web browser using the HTML5 Canvas API in Haskell. Haxl [23] is a Haskell library and EDSL for efficient access of remote data sources, and is also shallowly embedded. These EDSLs share the characteristic of ease of development, since given their shallow embedding, their syntax is close to idiomatic Haskell. They also allow quick turnaround, as intermediate results of computations can be observed on the host, allowing ease of debugging.

The lower right quadrant of the table lists examples of EDSL's which are deeply embedded and include code generation. Kansas Lava [12] is an EDSL for hardware entities, and is able to generate VHDL. Felsdspar [1][2] is an EDSL for describing digital signal processing algorithms, and is able to generate C language code from the EDSL. Ivory [8] is an EDSL that is designed as a language for safe systems level programming, and also generates C language code. All of these EDSL's have the characteristics of better performance and better resource utilization than shallow EDSL's, due to the ability to generate code in a low level language.

The lower left quadrant in the table is the worst of both worlds. With EDSLs in this quadrant, the user must write in a harder to use deep language, and live with lower performance and suboptimal resource utilization, since code is not generated in a lower level language. One use we have found for languages in this quadrant is debugging the development of a Deep EDSL prior to code generation.

The upper right quadrant is where systems that combine ease of use and optimized performance may be found. EDSLs in this quadrant are able to be written in the easier to use shallow embedding, and have the better performance and resource utilization characteristics of code generation. Also, if the same shallow code can be used with native execution/interpretation, as well as code generation, the user can first prototype with native execution or interpretation, and then deploy with code generation. It is a solution that enables programming in this quadrant that we explore in this paper. Conal Elliott's work in compiling to categories [7] is another method for developing a programing system that exhibits attributes of this quadrant, combining ease of use with analysis and optimization.

In this paper, we make the following contributions:

- We provide a method of transforming a shallow monadic EDSL to a deep monadic EDSL using a worker-wrapper transformation and a set of transformation rules (Sections 4 and 5).
- We also provide a method for transforming tail recursive functions in the same monadic EDSL into an iterative structure once again using transformation rules (Section 6).
- To automate the transformations listed above, we have developed a GHC compiler plugin. This plugin allows the user to write shallow, tail recursive EDSL code, and have it transformed into deep, iterative EDSL code as part of the compilation process (Section 7).

2 Haskino

Figure 1 illustrates the system we have developed to advance the use of functional languages on the Arduino, known as Haskino [16]. The figure shows the capabilities that have been developed in each of three steps of research.

In step 1, we developed a shallow EDSL, as well as a deep EDSL, both written in Haskell [17]. They allow the end user to write a program on the host, while the computations specified by the program are executed on the Arduino using a firmware interpreter. The shallow EDSL allows the user to interactively program the Arduino, with intermediate results being returned to the host computer connected by USB cable. This interactive setup makes debugging and prototyping of new code and hardware much easier. The deep EDSL provides a way of outsourcing entire groups of commands and control-flow idioms to the Arduino. This allows a user's Haskell program to store a bytecode program on the board, then step back and let it run. Both of these EDSL methods use the remote monad design pattern to provide the key capabilities.

For step 2 of the research, Haskino was modified to use the same monadic code that is executed with the interpreter, but instead compile it into C code [18]. That C code may then be compiled and linked with a small runtime, to allow standalone operation of an executable with a smaller size than the interpreted code. This smaller size allows more complex programs to be developed and executed within the limited resources of the Arduino. In addition, the second stage of Haskino research added the concept of multi-threaded operation to the system. Programming embedded microcontrollers often requires the scheduling of independent threads of execution, specifying the interaction and sequencing of actions

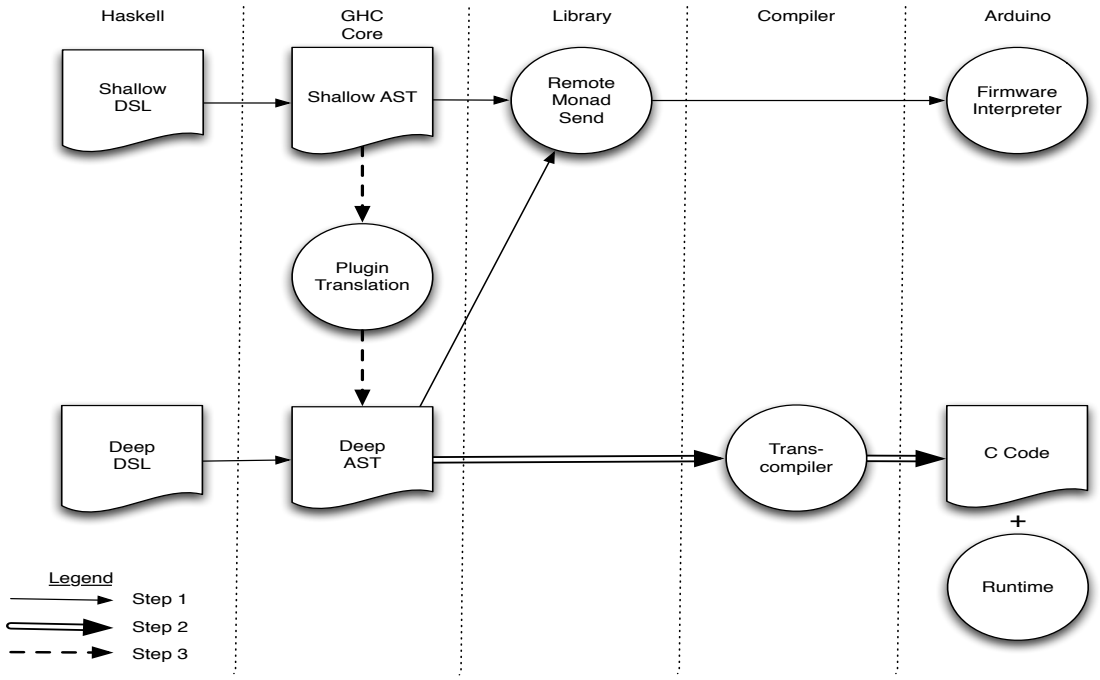

Figure 1. Haskino Overview

in the multiple threads. We add thread scheduling and inter-thread communication concepts to the EDSLs. In addition, we added scheduling and communication capabilities enabling those concepts to both the firmware interpreter and the small run time.

In step 3 of the research, detailed in this paper, we explore enabling the user to write programs once in the shallow EDSL, and then have the code automatically translated to the deep EDSL by the Haskino system. The syntax of the shallow EDSL is much closer to standard Haskell than the syntax of the deep EDSL, using Haskell operators and control flow mechanisms. In addition, we would like for the user to be able to use the recursive style of iteration that is normally used in functional programming, as opposed to imperative programming's traditional `for` and `while` structures. We would like the program translation to be able to translate tail recursive functions written in the shallow DSL into the `iterateE` structure used in the deep EDSL. To enable these translations, we have developed a plugin for the GHC compiler which is able to perform the translations using principled methods. With the translation system, the user is able to write the code once and use it with both the interpreted, interactive system as well as the compiled, efficient system.

3 Worker-Wrapper Transformations

The worker/wrapper transformation [13, 20] is a transformation that converts a computation of one type into a computation of another type (worker), wrapped by a function that converts between the types of the two computations (wrapper). These type of transformations are correctness preserving, and have been used in compilers and other applications for many years.

Assuming a function *f*, which has a the following form:

$$f = body$$

The right hand side, *body*, may have recursive calls to *f*. We can then replace the body with the wrapper function, *wrap* applied to the worker function, *work*. The worker function itself is an application of the un-wrapper function, *unwrap* to the body of the original function.

$$f = wrap\ work$$
$$work = unwrap\ body$$

Where the body function is of type B, and the work function is of type A, the types of the worker/wrapper transformation are illustrated in Figure 2.

Applying this principle to our desired translations of shallow to deep EDSLs, and using the terminology of data representation [19], we define the `rep` function which is the un-wrapper which moves from our normal Haskell types to the `Expr` representation, and `abs` which converts from the

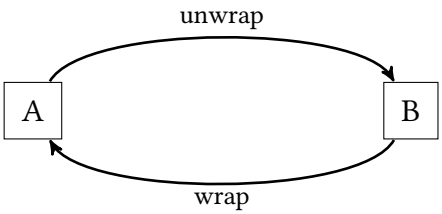

Figure 2. Worker-Wrapper Transformation

representation back to the abstract type. This is illustrated in Figure 3.

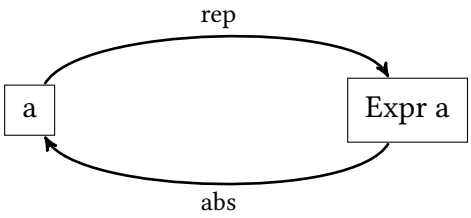

Figure 3. Expression Transformation

These conversions between the *abstract* type `a` and the *concrete* type `Expr a` depend on the worker-wrapper assumption that:

$$wrap \circ unwrap = id_A$$

Stating this in our DSL terms:

$$abs \circ rep = id_A$$

In the context of our expression language, that means that if we take a literal value in a base Haskell type such as Word8, move it the expression language with the `rep` function, then evaluate the resulting expression with the `abs` function, we will get the original value back.

4 Shallow to Deep Transformation

Our shallow to deep transformation of a program written in a monadic EDSL uses a worker-wrapper based transformation to move from a shallowly to a deeply embedded language. We start with a basic example in the shallowly embedded Haskino language to demonstrate how our transformations work. The following simple Haskino program reads the value of two digital inputs, which represent the state of two buttons, and then outputs a value to a digital output. The output will light a LED if either button one or button two is pressed. The program runs in an infinite loop (we will later show in Section 6 how this can be written in a recursive style and transformed into the loop primitive), with a 1 second delay between each loop.

```
let button1 = 2
let button2 = 3
let led = 13
loop $ do
    a <- digitalRead button1
    b <- digitalRead button2
    digitalWrite led (a || b)
    delayMillis 1000
```

The primitives used in the example are shallow commands and procedures in the Haskino Remote Monad [15] based DSL. They have the following types:

```
loop :: Arduino () -> Arduino ()
digitalRead :: Word8 -> Arduino Bool
digitalWrite :: Word8 -> Bool -> Arduino ()
delayMillis :: Word8 -> Arduino ()
```

With the shallow version of Haskino, the results of each of the computations (in this example, the value of the button state in `digitalRead`) are returned to the host computer and used in computations which will be sent as parameters of future commands (the value of the LED state in `digitialWrite` in this example).

The Haskino language also has a deeply embedded form, where parameters to the commands and procedures are not native Haskell types, but instead are values in an expression type, `Expr`. The type class `ExprB` is used to specify Haskell base types which may be lifted into the `Expr` type. The deeply embedded version of the language allows all of the computations to take place remotely on the Arduino, either as a stored program running in the Haskino interpreter, or translated to C, and compiled to assembly language to run directly on the Arduino.

The deep versions of the Arduino primitives used in our example have the following types:

```
loopE :: Arduino () -> Arduino ()
digitalReadE :: Expr Word8 -> Arduino (Expr Bool)
digitalWriteE :: Expr Word8 -> Expr Bool ->
                    Arduino ()
delayMillisE :: Expr Word8 -> Arduino ()
```

Using these deep primitives, we could write a deeply embedded version of our shallowly embedded example, which would have the following form:

```
let button1 = 2
let button2 = 3
let led = 13
loopE $ do
    a <- digitalReadE (lit button1)
    b <- digitalReadE (lit button2)
    digitalWriteE (lit led) (a ||* b)
    delayMillisE (lit 1000)
```

The `lit` operations lift a basic Haskell type into the Expr expression type, and the `||*` operator is the logical or operation between two values of the `Expr Bool` type.

Writing even this simple example in the deeply embedded style presents challenges to the programmer, as opposed

to the shallowly embedded style, which is more idiomatic Haskell. The overhead to writing in the deeply embedded style becomes even greater when using conditionals and iteration. We would prefer that the programmer be able to write in the shallowly embedded style, and let the compiler transform it to the deeply embedded style automatically. For now, we will show how this simple example may be written in the shallow version, and automatically converted to the deep version. We will deal with conditionals and iteration later in Section 5 and Section 6.

In the following steps in the transformation, we will omit the initial let expressions, and concentrate on the loop body. First, we will de-sugar the do notation, and get the following form:

```
loop (
  digitalRead button1 >>=
    (\ a -> digitalRead button2 >>=
      (\ b -> digitalWrite led (a || b))) >>
        delayMillis 1000)
```

In the first step of the transformation, we will convert each of the shallow commands and procedures into their deep versions, inserting worker-wrapper `abs` and `rep` function calls (as we described in Section 3) to maintain the types of the overall computation. The `rep` function moves a value from the basic Haskell type to one of the Expr type, and the abs function has the opposite effect, moving a value from a Expr type to a basic Haskell type. The `rep` is equivalent to the `lit` function of the `ExprB` type class, and may remain in the transformed code, while the `abs` function should never actually be evaluated in the transformed code.

```
rep :: ExprB a => a -> Expr a
rep w = lit w

abs :: Expr a -> a
abs _ = error "Internal error: abs called"
```

In the following descriptions of transformations, we will describe the transformations in the style of GHC rewrite rules [21]. In some cases the rules given would not be valid for GHC, as the left hand side is not a function application, however the syntax of the rules provides a convenient representation notation for our transformations.

The first transformation is of procedures, which return values. Each use of a shallow procedure `proc` will be transformed to use a deep version, `procE`. These shallow and deep versions have the forms:

```
proc :: a1 -> ... -> an -> Arduino b
procE :: Expr a1 -> ... -> Expr an ->
         Arduino (Expr b)
```

The term `proc` represents a generic shallow procedure, and `procE` represents a generic deep procedure. A specific transformation will be needed for each of the actual procedures in the DSL, but we use this generic procedure to show the form of the transformation. This transformation is achieved using the following rule.

```
forall (arg1 :: a1) ... (argn :: an).
proc arg1 .. argn
  =
abs <$> (procE (rep arg1) ... (rep argn))
```

Transformations for commands are simpler, because they return unit, they do not require the fmap application of the abs operator. Applying this transformation step to our example program, we get the following:

```
loopE (
  abs <$> digitalReadE (rep button1) >>=
    (\ a -> abs <$> digitalReadE (rep button2) >>=
      (\ b -> digitalWriteE (rep led)
                        (rep (a || b)))) >>
        delayMillisE (rep 1000))
```

Now that we have the worker-wrapper operators placed in our code, the next step in the transformation involves moving the rep operators inside of expressions, transforming the shallow functions over standard Haskell types into deep functions over types in the Expr data type. We refer to these transformations as "rep push" operations, pushing the rep operators to the interior of expressions. The instance of this type of transformation used in our simple example is for the boolean or operator, and the rule for the transformation is:

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

After applying the `rep` push transformation rule to the example, the expression is transformed from a `Bool` type into a `Expr Bool` type as shown below.

```
loopE (
  abs <$> digitalReadE (rep button1) >>=
    (\ a -> abs <$> digitalReadE (rep button2) >>=
      (\ b -> digitalWriteE (rep led)
                    ((rep a) ||* (rep b)))) >>
        delayMillisE (rep 1000))
```

Now that we have moved the `rep` functions inside of expressions, we can apply the next rule of the transformation, starting to move the abs operators closer to the rep operators to achieve fusion. This rule is a variant of the third monad rule, and has the form:

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

Applying this monadic rule to the example, we move the `abs` operators through the two monadic binds, changing them to a composition of the continuation with the `abs`.

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a -> digitalReadE (rep button2) >>=
      (\ b -> digitalWriteE (rep led)
                    ((rep a) ||* (rep b))) . abs
        ) . abs >>
      delayMillisE (rep 1000))
```

Having moved the abs operators through the binds, with the next rule we would like to move the abs operators inside of the lambdas. The transformation to do this has the following form:

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

The notation `f[x]` represents the usage of the binding x somewhere inside the function f. When this rule is applied to the example, two let expressions are inserted into the lambda expressions, as show below:

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a' -> let a = abs a' in
      digitalReadE (rep button2) >>=
        (\ b' -> let b = abs b' in
          digitalWriteE (rep led)
                        ((rep a) ||* (rep b)))) >>
      delayMillisE (rep 1000))
```

The let expressions may then be eliminated by replacing instances of a and b in the body of the lambdas with `(abs a')` and `(abs b')` respectively. This will result in:

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a' -> digitalReadE (rep button2) >>=
        (\ b' -> digitalWriteE (rep led)
                        ((rep (abs a')) ||*
                         (rep (abs b'))))) >>
        delayMillisE (rep 1000))
```

Now, with the `rep` and `abs` applications correctly positioned, one final simple transformation is required. The rep-abs combinations may be fused by the following rule.

```
forall x.
rep(abs(x))
  =
x
```

At this point the transformed code is equivalent to the hand written deeply embedded code, since `rep` is equivalent to `lit`, and we have achieved our goal with the transformation.

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a' -> digitalReadE (rep button2) >>=
        (\ b' -> digitalWriteE (rep led)
                        ( a' ||* b'))) >>
        delayMillisE (rep 1000))
```

The transformations described in the example in this section, and which are implemented in our plugin (Section 7), currently cover monadic code written with the higher level Haskell functions >>= and >>. They do not handle other higher level Haskell monadic functions such as `mapM`. The worker-wrapper transformation techniques used in this section could be extended to define rules for transforming instances of mapM and other higher order functions.

5 Conditionals

Conditionals in deeply embedded DSLs normally take the form of functions over three arguments, one for the boolean test, and one each for the then and else branch of the conditional. Writing the conditionals in this form, as opposed to the normal Haskell if-then-else form, is another case where writing code for a deeply embedded DSL is inconvenient. Our transformations once again allow the program author to write in a shallowly embedded DSL form, like standard Haskell coding, and have the program automatically transformed to the deep EDSL form.

There are two types of conditionals which must be transformed. The first of these are conditionals where the then and else expressions are of the main data type of the EDSL. Once again, using our example of the Haskino language, these are terms of the `Arduino` monad type. In the Haskino language, this type of conditional function has the following type:

```
ifThenElseE :: ExprB a => Expr Bool ->
                          Arduino (Expr a) ->
                          Arduino (Expr a) ->
                          Arduino (Expr a)
```

We will use another small example code section which deals with three button inputs and two LED outputs to demonstrate the conditional transformation:

```
a <- digitalRead button1
b <- if a
     then do
        digitalWrite led1 True
        digitalRead button2
     else do
        digitalWrite led2 True
        digitalRead button3
```

The main transformation of the conditional is similar to the command and procedure transformations we performed in Section 4. The rule syntax for the transformation is as follows:

```
forall (b :: Bool) (m1 :: ExprB a => Arduino a)
                   (m2 :: ExprB a => Arduino a).
if b then m1 else m2
  =
abs <$> ifThenElseE (rep b) (rep <$> m1)
                            (rep <$> m2)
```

Applying this rule to our example, after also applying the command and procedure transformations from Section 4, the

shallow code containing the conditional is transformed into the following:

```
a <- digitalRead button1
b <- abs <$> ifThenElseE (rep a)
              (rep <$> do
                digitalWrite (rep led1) (rep True)
                digitalRead button2)
              (rep <$> do
                digitalWrite (rep led2) (rep True)
                digitalRead button3)
```

At this point, two more manipulation rules need to be added to the transformation toolbox. The first is a rule which will push the `fmap` application of `rep` through the monadic binds in the then and else branches. It has the following form:

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

After the application of this rule, along with the rules described in Section 4, our conditionals example is transformed to the following:

```
a' <- digitalReadE button1
b' <- ifThenElseE (a')
        (do
           digitalWriteE (rep led1) (rep True)
           rep <$> (abs <$> digitalReadE
                                (rep button2)))
        (do
           digitalWriteE (rep led2) (rep True)
           rep <$> (abs <$> digitalReadE
                                (rep button3)))
```

Now the final rule required is the `fmap` analog to the rep-abs fusion rule we used earlier in Section 4, which will fuse the `rep` and `abs` functions in the then and else branches.

```
forall (m :: Expr a => Arduino a).
rep <$> (abs <$> m)
  =
m
```

The other form of conditional found in many deeply embedded DSLs is a conditional over the expression language. In the Haskino language, this conditional has the following type:

```
ifB :: ExprB a => Expr Bool ->
                  Expr a -> Expr a -> Expr a
```

Transformation of an if-then-else expression written in a shallowly embedded form to this deeply embedded conditional requires only one transformation rule, as shown below. Following this application rule, the rep-push rules described in Section 4 may be used to further reduce the expressions in the boolean test, as well as the then and else branches of the expression.

```
forall (b :: Bool) (t :: ExprB a => a)
                   (e :: ExprB a => a).
if b then t else e
  =
abs $ ifB (rep b) (rep t) (rep e)
```

6 Iteration and Recursion

An Arduino C programmer would use `for` or `while` loops for programming iteration, however, a Haskell programmer would use recursion for the same task. The Haskino deep EDSL provides a iterateE structure for interation, but as our goal is to provide relatively idiomatic Haskell syntax to the programmer, using it is unsatisfying. Instead, we would like to be able to translate tail recursive functions in the shallow EDSL into functions using the iterateE structure automatically, as we have done with conditionals and the other shallow components of the DSL.

6.1 First Recursion 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
```

We would like to transform recursive functions of the type `Expr a -> Arduino(Expr b)` into functions which use an imperative iteration loop. We enable the transformation by creating a data type, `Iter` which indicates if on a specific iteration of the loop, the function should return (it is "Done"), or if it should perform the computation associated with the next iteration of the loop (it needs to "Step").

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

We also define a Haskell function which performs the iterative loop. This function will not be used in the final implementation, but is defined to allow us to demonstrate the transformation method in the shallow version of the DSL. The iteration function, `iterLoop`, takes the initial value of the input argument, and a function which is able to perform a single step of the iteration that returns either a Step value of the input type, or a Done value of the output type.

```
iterLoop :: a -> (a -> Arduino (Iter a b)) ->
            Arduino b
iterLoop iv stepF = do
    result <- stepF iv
    case result of
        Step va -> iterloop va stepF
        Done vb -> return vb
```

We start the transformation by adding a wrapper function to insert our `iterLoop` function. The worker function, `blink` is formed by applying the Done constructor to the body of the original function, in which we have removed the pattern matching notation, replacing it with conditional notation.

```
blink :: Word8 -> Arduino ()
blink a = iterLoop blinkI a

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

We now apply the first rule of our recursion transformation to the function, which is used to move the Done constructor through any conditionals:

```
forall f x.
    Done <$> if b then f else g
        =
    if b then (Done <$> f) else (Done <$> g)
```

Applying this rule to our function gives:

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

We now introduce the other basic rule of our recursive transformation to move the Done constructor call to the end of the bind chain.

```
forall f g.
    Done <$> (f >>= g)
        =
    f >>= \ x -> Done <$> g x.
```

Applying this rule repeatedly we obtain:

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

Finally, in the branches where the recursive function call is present, we can apply the following rule to eliminate the recursive call, `f`, instead inserting the Step constructor.

```
forall x.
    Done <$> (f x)
        =
    Step <$> (return x)
```

With our example, and applying the rule, it then becomes:

```
blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI t = if (t == 0)
              then Done <$> return ()
              else do
                  digitalWrite led True
                  delayMillis 1000
                  digitalWrite led False
                  delayMillis 1000
                  Step <$> (return (t-1))
```

Finally, we use the following rule involving the `Done` constructor, and an equivalent one for the `Step` constructor.

```
forall x.
    Done <$> (return x)
        =
    return (Done x)
```

This moves the constructor inside of the return, and leaves us with the final transformed version:

```
blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI t = if (t == 0)
              then return (Done ())
              else do
                  digitalWrite led True
                  delayMillis 1000
                  digitalWrite led False
                  delayMillis 1000
                  return (Step (t-1))
```

6.2 Translating to Haskino Iteration

The example transformation of the last section was demonstrated on the shallow version of the DSL for clarity. We would now like to replace the `iterLoop` function which is written in Haskell, with the Haskino iteration primitive. This primitive has the following type:

```
iterateE :: Expr a ->
            (Expr a -> Arduino (ExprEither a b)) ->
            Arduino (Expr b)
```

The Haskino expression language also has an `ExprEither` type, which will be used instead of the `Iter` type, and is defined as:

```
data ExprEither a b where
    ExprLeft   :: (ExprB a, ExprB b) =>
                  Expr a -> ExprEither a b
    ExprRight  :: (ExprB a, ExprB b) =>
                  Expr b -> ExprEither a b
```

As the iterateE and ExprEither are defined in the deep version of Haskino, it is preferable to do the recursive transforming after first transforming the example from the shallow language to the deep using the transformation from Section 4. Doing so gives us the following for the example from the previous section:

```
blinkE :: Expr Word8 -> Arduino (Expr ())
blinkE t =
        ifThenElseE (t ==* rep 0)
             (return (rep ()))
             (do
                digitalWriteE (rep led) (rep True)
                delayMillisE (rep 1000)
                digitalWriteE (rep led) (rep False)
                delayMillisE (rep 1000)
                blinkE  (t - (rep 1))
```

We now can use the method we demonstrated in the previous section, using `iterateE` instead of `iterLoop`, and the `ExprEither` type instead of the Iter type. We also need to replace the conditional used in the function, as `ifThenElseE` is only defined over one type, and instead we will use the `ifThenElseEither` which is defined over two types.

```
ifThenElseEither :: (ExprB a, ExprB b) =>
                    Expr Bool ->
                    Arduino (ExprEither a b) ->
                    Arduino (ExprEither a b) ->
                    Arduino (ExprEither a b)
```

Applying the recursion transformation method, using the `ExprLeft` constructor in place of the `Step` constructor, and `ExprRight` in place of `Done` we get:

```
blinkE :: Expr Word8 -> Arduino (Expr ())
blinkE t = iterateE t blinkEI

blinkEI :: Expr Word8 ->
                Arduino (ExprEither Word8 ())
blinkEI t =
        ifThenElseEither (t ==* rep 0)
             (return (ExprRight (rep ())))
             (do
               digitalWriteE (rep led) (rep True)
               delayMillisE (rep 1000)
               digitalWriteE (rep led) (rep False)
               delayMillisE (rep 1000)
               return (ExprLeft (t - (rep 1)))
```

6.3 Second Recursion Example

As a second example, we present a transformation of recursion which demonstrates both a recursive function which returns a non-unit value, and the ability to greatly simplify a deep EDSL function by being able to write it in the shallow EDSL.

A common peripheral used on the Arduino is a small LCD display, and some of these display units also have a set of five buttons that may be used to indicate Up, Down, Left, Right, and Select. A press of one of these buttons is detected by the user program by reading a 16 bit analog input, where each single button press is denoted by a range of values. The shallow version of a recursive function which waits for one of the keys to be pressed, and returns an 8 bit unsigned integer corresponding to the button pressed is shown below:

```
analogKey :: Arduino Word8
analogKey = do
    v <- analogRead button2
    case v of
         | v < 30  -> return KeyRight
         | v < 150 -> return KeyUp
         | v < 350 -> return KeyDown
         | v < 535 -> return KeyLeft
         | v < 760 -> return KeySelect
     _             -> analogKey ()
```

The following is the resulting deep version after shallow to deep and recursive transformations:

```
analogKeyE :: Arduino (Expr Word8)
analogKeyE = analogKeyE' (lit ())

analogKeyE' :: Expr () -> Arduino (Expr Word8)
analogKeyE' t = iterateE t analogKeyE'I

analogKeyE'I :: Expr () ->
                Arduino (ExprEither () Word8)
analogKeyE'I _ = do
  v <- analogReadE button2
  ifThenElseEither (v <* 30)
    (return (ExprRight (lit KeyRight)))
    (ifThenElseEither (v <* 150)
      (return (ExprRight (lit KeyUp)))
      (ifThenElseEither (v <* 350)
        (return (ExprRight (lit KeyDown)))
        (ifThenElseEither (v <* 535)
          (return (ExprRight (lit KeyLeft)))
          (ifThenElseEither (v <* 760)
            (return (ExprRight (lit KeySelect)))
            (return (ExprLeft (lit ()))))))))
```

Our method of transformation from Sections 6.1 and 6.2 works for functions with one argument, but not with functions with zero arguments as in this example. To transform functions of this type require us to first transform the function into one that takes a parameter of type `Expr ()` which is not used in the body of the function, but allows us to use the

iterateE construct, and the ExprEither type, both of which are parameterized over two types.

With this example, we can see the advantage of using the pattern matching notation in the shallow version, as opposed to the more verbose deep notation.

7 Plugin Architecture and Implementation

The shallow to deep transformations, as well as the recursive transformations in our system, are implemented using the GHC Plugins mechanism [11]. Our plugin manipulates the Haskell module being compiled through a series of Core to Core passes, where Core is GHC's intermediate language[24].

A Haskell Module's top level ModGuts data structure is carried throughout all phases of the compiler, including plugin passes. This data structure contains not only the Core of the module under compilation, but also a global reader environment of all in-scope symbols, GHC transformation rules, information about other modules imported to the one under compilation, and other information useful to the compiler pass. Each pass of a GHC plugin is defined as the following type:

```
ModGuts -> CoreM ModGuts
```

Each of the passes in our plugin transforms the list of Core Bind's, which are part of the ModGuts data type, into another list of Core Bind's in the returned ModGuts. The plugin operates on Bind's that are of the type of the DSL's Monad, which in our case study is one of type `Arduino a`. By the time that the compiler has translated native Haskell into Core, Bind's are separated into recursive (`Rec`) and non-recursive (`NonRec`) Bind's. The passes associated with the shallow to deep transformation operate on both Bind's constructed with `NonRec` and those constructed with `Rec`, while those associated with the tail recursion transformation operate only on the Bind's constructed with `Rec`.

The plugin has been designed to be customized for other monadic EDSLs, and not be used just for our case study EDSL of Haskino. The types of the DSL monad and expression types are specified in a single module as Template Haskell names, allowing the plugin to be customized quickly for a new EDSL based on the remote monad monadic structure. Similarly, tables of EDSL primitives and rules components are used in several of the passes to allow for EDSL customization, and they will be described in more detail later in this section. Finally, as the plugin has been developed, we have built up the basis for a plugin toolkit which is designed to be lighter weight than such tools as Hermit [10]. For example, we have generalized and made into a utility the routines from Hermit which are used to look up Core dictionaries, which frequently need to be generated as part of the transformation process.

The plugin operates on a per module basis, transforming all functions of the DSL type present in the module. GHC plugins may be invoked by specifying the `-fplugin` flag with the plugin name either on the command line, or in a compiler directive within the file. This allows us to specify on a module by module basis if the transformations will be done. Using this method, a file may still be written directly in the Deep EDSL without transformations, or in the Shallow EDSL with transformations. The can be useful for regression testing during development of the plugin, allowing the results of the transformation to be compared the the native Deep code.

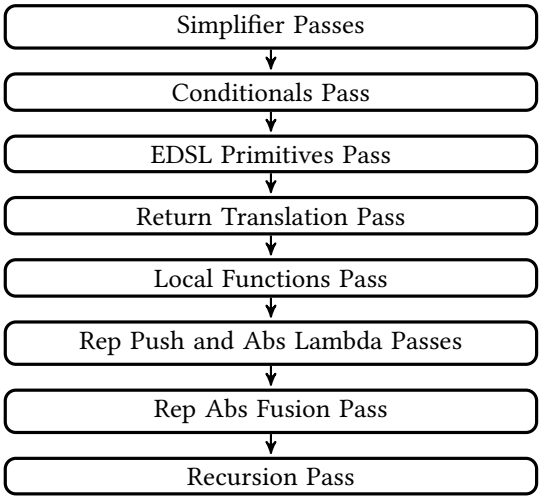

Figure 4. Structure of Transformation Plugin Passes

The structure of the passes implemented by the GHC plugin is shown in Figure 4. Each of the passes of the plugin are described in one the following sections. The current ordering of the passes is required for the passes to function as written, and has been chosen to optimize the amount of code required for each pass. However, the optimal ordering as well as other optional orders, are still under investigation as part of our ongoing research.

7.1 Simplifier Passes

The first pass executed by the plugin is a pass to run the GHC simplifier, without any inlining, rule rewriting, or eta-expansion.

This pass is run to complete any inlining of functions that may be have been done in the GHC compiler before it passes the Core for the module to our plugin. We found that some functions that were inlined would be left in the form of `(\x -> F[x])(y)`, and running the simplifier pass will perform the function application, and leave the Core in a standard form for transformation by the rest of the plugin.

The second simplifier executed by the plugin is a pass which removes the Haskell application operator `$`, which is still present in the Core that the plugin receives. Replacing this operator with a standard function application reduces the number of rules that are required for subsequent passes.

7.2 Conditionals Pass

The Conditionals Pass transforms standard Haskell if-then-else expressions into the DSL's embedded if-then-else constructs. The pass searches the Core for two armed Case expressions with arms of False and True, which return a type of Arduino a, and transforms them into the EDSL's `IfThenElseE` primitives according to the rules defined in Section 5. Note, that this transformation will also transform the syntax of Haskell Case pattern matching with guards of the type shown in the analogKey example in Section 6.3.

Similarly, it looks for two armed Case expressions with arms of False and True, which return a type in the EDSL's expression type class, and transforms them into the EDSL's `ifB` primitive.

7.3 EDSL Primitives Pass

The EDSL Primitives Pass translates the EDSL primitives (in the case of a Remote Monad based EDSL, commands and procedures are the primitive data types) from their shallow form to their deep form, as was shown by the rules in Section 4.

This pass performs the equivalent of the rules using a Core-to-Core pass. It does this using a table of pairs of Template Haskell names, the first element of the pair being the shallow version of the primitive, and the second element being the deep version of the primitive. The pass recursively searches the Core for function applications of the first element (in Core syntax, function application is represented by the `App` data structure), and replaces it with an application of the second. The pass compares the types of the two versions, and performs the translation by adding application of `rep` and `abs` functions as needed. For both commands and procedures, it applies a `rep` function to each of the arguments of the primitive, and for procedures, it also applies an `abs` to the return value of the primitive using the fmap function, since the return is of a monadic type.

7.4 Return Translation Pass

For monadic based DSLs, such as Haskino, instances of `return` functions need to be transformed just as the EDSL primitives are. This pass transforms the returns with the equivalent of the following rule:

```
forall (x :: ExprB a => a)
  return x
    =
  abs <$> return (rep x)
```

This transformation is equivalent to the transformation of procedures used in the EDSL Primitives Pass.

7.5 Local Functions Pass

We had two options for handling local function definitions in the module being compiled. As running most deeply embedded DSLs consists of inlining those functions, the first option is to simply inline any applications of those functions in other functions inside of the module.

However, we are planning on adding Lambda expressions to the Haskino Deep EDSL in the next phase of research, and therefore we wanted a method that does not inline everything in this pass of the transformation. Instead, we wanted to translate all local functions that return type `Arduino a` in place.

This pass replaces the shallow function body with a call of a deep version of the function, as in the following example:

```
myRead :: Word8 -> Arduino Bool
myRead p = abs <$> (myReadDeep (rep p))

myReadDeep :: Expr Word8 -> Arduino Bool
```

What was the former body of the shallow version is transformed by this pass to become the new body of the deep version. Using this method, a module being transformed that is dependent on a previously transformed module will pass type checking during the initial GHC type checking phase, before the untransformed Core is given to the plugin, since both shallow and deep versions will be present in the previous module.

This pass also replaces applications of shallow functions, with applications of the deep functions. This means it applies a `rep` function to each of the arguments of the function, and an `abs` to the return value of the function using the fmap function, similar to how EDSL procedures are transformed.

7.6 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 worker-wrapper functions for possible fusion in the next pass.

The Rep Push pass is performed in the plugin in a similar method to the DSL primitive pass, in that it contains a table of pairs of the functions with the from and to functions to transform for each of the EDSL Expr language operations. (In the example in Section 4 discussing the Rep Push transformations, this pair consists of the `(||)` and `(||*)` functions). One of the sets of pairs is defined for each of the DSL's Expr operations, to move the operation from a shallow operation in basic Haskell types, to a deep operation using the Expr language operators.

The Abs Lambda pass performs the equivalent of rules described in Section 4. Two rule analogues push the `abs` function applications through monadic >>= and >> operators, and move the `abs` inside of lambdas. The second rule and the elimination of the let expression is done in one step, which has the form shown below.

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

The abs function composition is eliminated, the lambda argument x is renamed to x', it's type is changed to Expr a, and any occurrence of x in the body of the lambda is replaced with abs(x').

7.7 Rep Abs Fusion Pass

This pass fuses the rep and abs pairs that have been moved next to each other by the previous passes. It performs the equivalent of the two rep-abs fusion rules described in Section 4. After this pass is complete, there will be some applications of rep left in the Core, where it is required to lift literal basic Haskell values into the EDSL's Expr language. There also may be some applications of abs left in the Core, but due to Haskell's lazy evaluation, these will never be evaluated. This will occur when an EDSL procedure's return value is not bound to a lambda argument, but is instead used with the >> operator instead of the >>= operator. To simplify the generated Core, these unevaluated instances of abs are eliminated by a separate pass following the fusion pass.

7.8 Recursion Pass

The Recursion Pass transforms Deep EDSL tail recursive functions into the Deep EDSL's iterateE construct. It performs the equivalent of rules described in Section 6 with a Core-to-Core pass, and only operates on Core Bind's constructed with Rec. It transforms tail recursive Haskino functions with zero or one arguments. Recursive functions with larger number of arguments are currently flagged to the user as non-translatable. Transformation of these could be added to the pass, but would require the addition of tuples to the Haskino Deep EDSL.

A pass to detect functions with non-tail recursion, or with mutual tail recursion, and issue an error to the user that they are non-transformable, is planned for the next version.

8 Related Work

There have been several other efforts to blend shallow and deep EDSL's. Svenningsson and Axelsson [28] explored combining deep and shallow embedding. They used a deep embedding as a low level language, then extended the deep embedding with a shallow embedding written on top of it. Haskell type classes were used to minimize the effort of adding new features to the language.

Yin-Yang [22] provides a framework for DSL embedding in Scala which uses Scala macros to provide the translation from a shallow to deep embedding. Yin-Yang goes beyond the translation by also providing autogeneration of the deep DSL from the shallow DSL. The focus of Yin-Yang is in generalizing the shallow to deep transformations, and does not include recursive transformations.

Scherr and Chiba [26] proposed using load time implicit staging, as opposed to compile time mechanisms, as an alternative to deep embedding. Their prototype in Java allows the user to write in a shallow EDSL, then extracts expression semantics from Java bytecode at load time.

Forge [27] is a Scala based meta-EDSL framework which can generate both shallow and deep embeddings from a single EDSL specification. Embeddings generated by Forge use abstract Rep types, analogous to our EDSL's Expr types. Their shallow embedding is generated as a pure Scala library, while the deeply embedded version is generated as an EDSL using the Delite [3] framework.

Both Yin-Yang and Delite are built on top of Lightweight Modular Staging [25], a general purpose staging framework for developing deep EDSL's based on type directed transformations. Elliott developed GHC plugins [4][6] for compiling Haskell to hardware [5], using worker-wrapper style transformations equivalent to the abs and rep transformations used in the Haskino plugin. These plugins were later generalized to enable additional interpretations [7].

9 Conclusion and Future Work

Shallow EDSLs hosted in Haskell allow the programmer using the EDSL to write in relatively idiomatic Haskell, and provide a quick turnaround development environment. Deep EDSLs provide better performance and resource utilization by allowing code generation from the DSL's abstract syntax tree. We have shown, using as an example the Haskino EDSL being developed as part of our ongoing research, that automatic transformation of a shallow to deep EDSL can provide a combination of the benefits of both the shallow and deep EDSLs. With one set of source code, an EDSL user is provided with a quick turnaround, prototyping environment, and a higher performance, generated code system.

The current plugin is designed to be customized for use with different monadic EDSLs. In the future we would like to generalize the technique, allowing the plugin to be used with both monadic and non-monadic EDSLs. We would like to extend the rules and plugins used in the shallow to deep transformations to handle other higher level functions such as mapM. We also would like to generalize the tools and routines we used within the transformation plugin, to provide a framework or toolkit for writing such plugins.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1350901. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. We would also like to thank Conal Elliott for his ongoing discussions about reification and compiling embedded languages.

References

[1] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and Andras Vajdax. 2010. Feldspar: A domain specific language

for digital signal processing algorithms.. In *MEMOCODE'10*. 169–178.

[2] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. 2011. The design and implementation of Feldspar. In *Implementation and Application of Functional Languages*. Springer, 121–136.

[3] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, Washington, DC, USA, 89–100. DOI: http://dx.doi.org/10.1109/PACT.2011.15

[4] Conal Elliott. 2015. (2015). https://github.com/conal/lambda-ccc

[5] Conal Elliott. 2015. (2015). https://github.com/conal/talk-2015-haskell-to-hardware

[6] Conal Elliott. 2016. (2016). https://github.com/conal/reification-rules

[7] Conal Elliott. 2017. Compiling to categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 48 (Sept. 2017), 24 pages. DOI: http://dx.doi.org/10.1145/3110271

[8] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt free ivory. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. ACM, 189–200.

[9] Levent Erkok. 2014. Hackage package `hArduino-0.9`. (2014).

[10] Andrew Farmer, Neil Sculthorpe, and Andy Gill. 2015. Reasoning with the HERMIT: tool support for equational reasoning on GHC core programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. ACM, 23–34.

[11] GHC Team. 2016. *Glasgow Haskell Compiler User's Guide, Version 8.0.1*. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/

[12] Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmell, and Ed Komp. 2013. Types and Associated Type Families for Hardware Simulation and Synthesis: The Internals and Externals of Kansas Lava. *Higher-Order and Symbolic Computation* (2013), 1–20. DOI: http://dx.doi.org/10.1007/s10990-013-9098-7

[13] A Gill and G Hutton. 2009. The worker/wrapper transformation. *Journal of Functional Programming* (2009).

[14] Andy Gill and Ryan Scott. 2015. (2015). https://github.com/ku-fpg/blank-canvas

[15] Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. 2015. The remote monad design pattern. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. ACM, 59–70.

[16] Mark Grebe. 2017. (2017). https://github.com/ku-fpg/haskino

[17] Mark Grebe and Andy Gill. 2016. Haskino: A Remote Monad for Programming the Arduino. In *Practical Aspects of Declarative Languages*. Springer, 153–168.

[18] Mark Grebe and Andy Gill. 2017. Threading the Arduino with Haskell. In *Post-Proceedings of Trends in Functional Programming*.

[19] C. A. Hoare. 1972. Proof of Correctness of Data Representations. *Acta Informatica* 1, 4 (Dec. 1972), 271–281.

[20] SLP Jones and J Launchbury. 1991. Unboxed values as first class citizens in a non-strict functional language. *Conference on Functional Programming* (1991).

[21] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop 1*. 203–233.

[22] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-yang: Concealing the Deep Embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. ACM, New York, NY, USA, 73–82. DOI: http://dx.doi.org/10.1145/2658761.2658771

[23] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. In *International Conference on Functional Programming*. ACM, 325–337.

[24] Simon Peyton Jones and André L. M. Santos. 1998. A transformation-based optimiser for Haskell. *Science of Computer Programming* 32, 1–3 (1998), 3–47.

[25] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136. DOI: http://dx.doi.org/10.1145/1868294.1868314

[26] Maximilian Scherr and Shigeru Chiba. 2014. Implicit Staging of EDSL Expressions: A Bridge Between Shallow and Deep Embedding. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag New York, Inc., New York, NY, USA, 385–410. DOI: http://dx.doi.org/10.1007/978-3-662-44202-9_16

[27] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE '13)*. ACM, New York, NY, USA, 145–154. DOI: http://dx.doi.org/10.1145/2517208.2517220

[28] Josef Svenningsson and Emil Axelsson. 2013. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming*. Springer, 21–36.