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

Table 1. EDSL Options

<table>
<thead>
<tr>
<th>Shallow EDSL</th>
<th>Deep EDSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>hArduino</td>
<td>Kansas Lava</td>
</tr>
<tr>
<td>Blank Canvas</td>
<td>Feldspar</td>
</tr>
<tr>
<td>Haxl</td>
<td>Ivory</td>
</tr>
</tbody>
</table>

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...
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 runtime.

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:

\[
f = \text{body}
\]

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.

\[
f = \text{wrap work}
\]

\[
\text{work} = \text{unwrap body}
\]

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

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

\[ \text{wrap} \circ \text{unwrap} = \text{id}_A \]

Stating this in our DSL terms:

\[ \text{abs} \circ \text{rep} = \text{id}_A \]

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.

```haskell
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 digitalWrite 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:

```haskell
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 `rep` and `abs` 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.

```
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, inserting 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 an `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) >>=
  ((abs a) ||* (rep b)))
  =
  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`.
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:

\[ \forall (f :: \text{Arduino} \ a). (\lambda x \to f[x]) \cdot \text{abs} = (\lambda x' \to \text{let } x = \text{abs } x' \text{ in } f[x]) \]

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:

\[ \text{loopE (} \text{digitalReadE (rep button1)} \Rightarrow \text{digitalReadE (rep button2)} \Rightarrow \text{digitalWriteE (rep led)} ( (\text{rep } a) \lor* (\text{rep } b)) \cdot \text{abs} \Rightarrow \text{delayMillisE (rep 1000)} \) \]

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

\[ \text{loopE (} \text{digitalReadE (rep button1)} \Rightarrow \text{digitalReadE (rep button2)} \Rightarrow \text{digitalWriteE (rep led)} ((\text{rep } a') \lor* (\text{rep } b')) \Rightarrow \text{delayMillisE (rep 1000)} \) \]

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

\[ \forall x. \text{rep(abs(x))} = x \]

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

\[ \text{loopE (} \text{digitalReadE (rep button1)} \Rightarrow \text{digitalReadE (rep button2)} \Rightarrow \text{digitalWriteE (rep led)} ( a' \lor* b') \Rightarrow \text{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 \( \Rightarrow \) and \( \Rightarrow \). They do not handle other higher level Haskell monadic functions such as \text{mapM}. The worker-wrapper transformation techniques used in this section could be extended to define rules for transforming instances of \text{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 \text{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 \text{Arduino} monad type. In the Haskino language, this type of conditional function has the following type:

\[ \text{ifThenElseE} :: \text{ExprB} a \Rightarrow \text{Arduino} ((\text{Expr} a) \Rightarrow (\text{Expr} a) \Rightarrow (\text{Expr} a)) \]

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

\begin{verbatim}
loopE (digitalReadE (rep button1) >>=
  (\ a -> digitalWriteE (rep led) a . abs >>
    delayMillisE (rep 1000)) . abs >>
  delayMillisE (rep 1000))

a <- digitalRead button1
b <- if a
  then do
    digitalWrite led1 True
digitalRead button2
  else do
    digitalWrite led2 True
digitalRead button3
\end{verbatim}

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:

\[ \forall (b :: \text{Bool}) (m1 :: \text{ExprB} a \Rightarrow \text{Arduino} a) (m2 :: \text{ExprB} a \Rightarrow \text{Arduino} a).
  (\text{abs } b \Rightarrow (\text{ifThenElseE (rep b)} (\text{rep } m1) (\text{rep } m2)) \Rightarrow \text{delayMillisE (rep 1000)}) \]

Applying this rule to our example, after also applying the command and procedure transformations from Section 4, the
Rewriting a Shallow DSL using a GHC Compiler Extension

shallow code containing the conditional is transformed into the following:

\[
\begin{align*}
    a &\gets \text{digitalRead} \; \text{button1} \\
    b &\gets \text{abs} \; \langle\rangle \; \text{ifThenElse} \; (\text{rep} \; a) \\
    &\hspace{1em} (\text{rep} \; \langle\rangle \; \text{do} \\
    &\hspace{2em} \text{digitalWrite} \; (\text{rep} \; \text{led1}) \; (\text{rep} \; \text{True}) \\
    &\hspace{2em} \text{digitalRead} \; \text{button2}) \\
    &\hspace{1em} (\text{rep} \; \langle\rangle \; \text{do} \\
    &\hspace{2em} \text{digitalWrite} \; (\text{rep} \; \text{led2}) \; (\text{rep} \; \text{True}) \\
    &\hspace{2em} \text{digitalRead} \; \text{button3})
\end{align*}
\]

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

\[
\forall \; (f :: \text{Arduino} \; a) \; (k :: a \rightarrow \text{Arduino} \; b). \\
\text{rep} \; \langle\rangle \; (f \; \gg= \; k) \\
= \\
f \; \gg= \; \backslash \; x \rightarrow \text{rep} \; \langle\rangle \; k \; x.
\]

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

\[
\begin{align*}
    a' &\gets \text{digitalReadE} \; \text{button1} \\
    b' &\gets \text{ifThenElseE} \; (a') \\
    &\hspace{1em} (\text{do} \\
    &\hspace{2em} \text{digitalWriteE} \; (\text{rep} \; \text{led1}) \; (\text{rep} \; \text{True}) \\
    &\hspace{2em} \text{digitalReadE} \; (\text{rep} \; \text{button2}) ) \\
    &\hspace{1em} (\text{do} \\
    &\hspace{2em} \text{digitalWriteE} \; (\text{rep} \; \text{led2}) \; (\text{rep} \; \text{True}) \\
    &\hspace{2em} \text{digitalReadE} \; (\text{rep} \; \text{button3}) )
\end{align*}
\]

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

\[
\forall \; (b :: \text{Bool}) \; (t :: \text{ExprB} \; a \rightarrow a) \\
\text{if} \; b \; \text{then} \; t \; \text{else} \; e \\
= \\
\text{abs} \; \text{ifB} \; (\text{rep} \; b) \; (\text{rep} \; t) \; (\text{rep} \; e)
\]

\section{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 an iterateE structure for iteration, 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.

\subsection{First Recursion Example}

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

\[
\begin{align*}
    \text{led} &= 13 \\
    \text{button1} &= 2 \\
    \text{button2} &= 3 \\
    \text{blink} :: \text{Word8} \rightarrow \text{Arduino} \; () \\
    \text{blink} \; 0 &= \text{return} \; () \\
    \text{blink} \; t &= \text{do} \\
    &\hspace{2em} \text{digitalWrite} \; \text{led} \; \text{True} \\
    &\hspace{2em} \text{delayMillis} \; 1000 \\
    &\hspace{2em} \text{digitalWrite} \; \text{led} \; \text{False} \\
    &\hspace{2em} \text{delayMillis} \; 1000 \\
    &\hspace{2em} \text{blink} \; \langle\rangle \; (t-1)
\end{align*}
\]

We would like to transform recursive functions of the type \(\text{Expr} \; a \rightarrow \text{Arduino} \; (\text{Expr} \; b)\) into functions which use an imperative iteration loop. We enable the transformation by creating a data type, \(\text{Iter}\) 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").

\[
\begin{align*}
    \text{data} \; \text{Iter} \; a \\
    &= \text{Step} \; a \\
    \mid \text{Done} \; b
\end{align*}
\]

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, \(\text{iterateLoop}\), 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 \text{Step} value of the input type, or a \text{Done} value of the output type.

\[
\begin{align*}
    \text{data} \; \text{Iter} \; a \\
    &= \text{Step} \; a \\
    \mid \text{Done} \; b
\end{align*}
\]
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)

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))

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:

6.2 Translating to Haskino Iteration
The example transformation of the last section was demon-
strated 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)

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
destructor, 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))

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 = 
  ifThenElseEither (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 ifThenElseEither 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:

\[ \text{ModGuts} \rightarrow \text{CoreM} \text{ModGuts} \]

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 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.

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.

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 of 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 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 \((x \rightarrow F[x])(y)\), 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.

![Figure 4. Structure of Transformation Plugin Passes](image-url)
We had two options for handling local function definitions. 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 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.

7.3 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 shown by the rules in Section 5. This transformation is equivalent to the transformation 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.

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.

7.4 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:

```haskell
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:

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

```haskell
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 is 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 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, 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.

```haskell
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 \( \text{Expr} \ a \), and any occurrence of \( x \) in the body of the lambda is replaced with \( \text{abs}(x') \).

7.7 Rep Abs Fusion Pass
This pass fuses the \( \text{rep} \) and \( \text{abs} \) pairs that have been moved next to each other by the previous passes. It performs the equivalent of the two \( \text{rep}-\text{abs} \) fusion rules described in Section 4. After this pass is complete, there will be some applications of \( \text{rep} \) left in the Core, where it is required to lift literal basic Haskell values into the EDSL’s \( \text{Expr} \) language. There also may be some applications of \( \text{abs} \) 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 \( \text{abs} \) 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 \( \text{iterateE} \) 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 \( \text{Rec} \). 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 \( \text{Rep} \) types, analogous to our EDSL’s \( \text{Expr} \) 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 \( \text{rep} \) 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 \text{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