Domain Specific Languages for Small Embedded Systems

Mark Grebe
Department of Electrical Engineering and Computer Science

The University of Kansas
April 27, 2018
Small Embedded Systems

- 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 Haskell directly on them.

- Embedded Domain Specific Languages (EDSL) provides an alternative.

- Using an EDSL a user is able to write in a high level, functional host language.

- Execution can be through either interpretation or compilation.
# Embedded Domain Specific Languages

<table>
<thead>
<tr>
<th>Shallow EDSL</th>
<th>Interpretation/Remote Execution</th>
<th>Code Generation/Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Examples</strong></td>
<td>Blank Canvas</td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>hArduino</td>
<td>Haskino</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>Ease of development</td>
<td>Ease of development</td>
</tr>
<tr>
<td></td>
<td>Quick turnaround</td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resource Optimization</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deep EDSL</th>
<th>Interpretation/Remote Execution</th>
<th>Code Generation/Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
<td>Debugging</td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kansas Lava</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Feldspar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ivory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Advantages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resource Optimization</td>
</tr>
</tbody>
</table>
Haskino Overview
Haskino Overview

Haskell
- Shallow DSL
- Shallow AST
- Plugin Translation
- Deep DSL
- Deep AST

GHC Core
- Shallow AST
- Remote Monad Send

Library
- Trans-compiler

Compiler
- Firmware Interpreter
- C Code
- Runtime

Arduino
Haskino Overview

- Haskell
  - Shallow DSL
  - Shallow AST
  - Plugin Translation
  - Deep DSL
  - Deep AST

- GHC Core
  - Shallow AST

- Library
  - Remote Monad Send

- Compiler
  - Trans-compiler
  - C Code
  - Firmware Interpreter

- Arduino
  - Runtime

- Deep AST

- GHC Core

- Library

- Compiler
Haskino Overview

Haskell

GHC Core

Library

Compiler

Arduino

Haskell

GHC Core

Library

Compiler

Arduino

Shallow DSL

Shallow AST

Remote Monad Send

Firmware Interpreter

Deep DSL

Deep AST

Plugin Translation

Trans-compiler

C Code

Runtime

Shallow DSL

Shallow AST

Remote Monad Send

Firmware Interpreter

Deep DSL

Deep AST

Trans-compiler

C Code

Runtime
Remote Monads

A remote **command** is a request to perform an action for remote effect, where there is no result value

\[
\text{digitalWrite} :: \text{Word8} \rightarrow \text{Bool} \rightarrow \text{Arduino} ()
\]
\[
\text{send} :: \text{ArduinoConnection} \rightarrow \text{Arduino} \ a \rightarrow \text{IO} \ a
\]

 GHCi> send conn $ digitalWrite 2 True
Arduino: LED on pin 2 turns on

A remote **procedure** is a request to perform an action for its remote effects, where there is a result value or temporal consequence

\[
\text{digitalRead} :: \text{Word8} \rightarrow \text{Arduino} \ \text{Bool}
\]

 GHCi> send conn $ digitalRead 3
Arduino: Returns the state of Pin 3
To demonstrate shallow Haskino syntax, I will use a simple Haskino example.

The example consists of two buttons and a LED and will light the LED if either button is pressed.

The shallow version of the example is:

```haskino
program :: Arduino ()
program = do
  let button1 = 2
      button2 = 3
      led = 13
  loop do
    a <- digitalRead button1
    b <- digitalRead button2
    digitalWrite led (a || b)
  delayMillis 100
```
Deep: Adding Expressions

The tethered shallow Haskino uses commands and procedures such as:

\[
\begin{align*}
\text{digitalWrite} & :: \text{Word8} \rightarrow \text{Bool} \rightarrow \text{Arduino ()} \\
\text{analogRead} & :: \text{Word8} \rightarrow \text{Arduino Word16}
\end{align*}
\]

To move to the deeply embedded version, we instead use:

\[
\begin{align*}
\text{digitalWriteE} & :: \text{Expr Word8} \rightarrow \text{Expr Bool} \rightarrow \text{Arduino (Expr ()}) \\
\text{analogReadE} & :: \text{Expr Word8} \rightarrow \text{Arduino (Expr Word16})
\end{align*}
\]
Expression Types

The Haskino EDSL provides `Expr a` parameterized over the following types, which are instances of the `ExprB` typeclass:

- `Word8`
- `Int8`
- `Bool`
- `Word16`
- `Int16`
- `Float`
- `Word32`
- `Int32`
- `[Word8]`

- Numeric operations include addition, subtraction, division, multiplications, comparisons, and conversion between numeric types.
- Boolean operations include `not`, `and`, and `or`. Integer operations include standard bitwise operations.
- `[Word8]` operations include append and element retrieval.
- Values are lifted into the `Expr` type by the `lit` function.
Conditionals

Conditionals become another data structure constructor when we move to the deep DSL:

```plaintext
button <- digitalWrite 2
if button
    then digitalWrite 2 True
else digitalWrite 3 True
```

```plaintext
button <- digitalReadE (lit 2)
ifThenElseE button (digitalWriteE (lit 2) (lit True))
    (digitalWriteE (lit 3) (lit True))
```
Transformations
Worker-Wrapper

- In general, these take a function 
  \( f = \text{body} \)
- And apply transforms such that
  \( f = \text{wrap work} \)
  \( \text{work} = \text{unwrap body} \)
- Moving between the A and B types.

- In our specific case, we move between \( a \) and \( \text{Expr } a \)
- \( \text{rep} \) is the equivalent of \( \text{lit} \), and \( \text{abs} \) corresponds to evaluation of the \( \text{Expr} \).
Shallow/Deep Translation

• Using worker-wrapper based transformations, the shallow DSL can be changed to the deep DSL.

• We automate this using a GHC plugin to do transformations in Core to Core passes.

```haskell
loop do
    a <- digitalRead button1
    b <- digitalRead button2
    digitalWrite led (a || b))
    delayMillis 100

loopE do
    a' <- digitalReadE (rep button1)
    b' <- digitalReadE (rep button2)
    digitalWriteE (rep led) ( a' ||* b'))
    delayMillisE (rep 100)
```
Translate the Primitives

Insert worker-wrapper ops by translating primitives of the form:

\[ a_1 \rightarrow \ldots \rightarrow a_n \rightarrow \text{Arduino } b \]

to ones of the form:

\[ \text{Expr } a_1 \rightarrow \ldots \rightarrow \text{Expr } a_n \rightarrow \text{Arduino } (\text{Expr } b) \]

```
loop (  
digitalRead button1 >>=  
  (\a -> digitalRead button2 >>=  
   (\b -> digitalWrite led (a || b))) >>=  
   delayMillis 100)
)
```

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

Translate the shallow operations to deep Expr operations:

\[ \text{rep} (\ x \ `\text{shallowOp} \ `y) \text{ transforms to (rep} x \ `\text{deepOp} \ `\text{rep} y) \]
where the types of shallowOp and deepOp are:

\[ \text{shallowOp :: a -> b -> c and deepOp :: Expr a -> Expr b -> Expr C} \]

\[
\text{loopE (}
\quad \text{abs <$> digitalReadE (rep button1) >>=}
\qquad (\ \text{\textbackslash} a \rightarrow \text{abs <$> digitalReadE (rep button2) >>=}
\qquad \quad (\ \text{\textbackslash} b \rightarrow \text{digitalWriteE (rep led) (rep (a || b)))}) \quad \text{>>}
\qquad \quad \text{delayMillisE (rep 1000)})
\]

\[
\text{loopE (}
\quad \text{abs <$> digitalReadE (rep button1) >>=}
\qquad (\ \text{\textbackslash} a \rightarrow \text{abs <$> digitalReadE (rep button2) >>=}
\qquad \quad (\ \text{\textbackslash} b \rightarrow \text{digitalWriteE (rep led) ((rep a ||* (rep b)))}) \quad \text{>>}
\qquad \quad \text{delayMillisE (rep 1000)})
\]
Move Abs Through Binds

Move the abs operations through the monadic binds

\((\text{abs} \ <$> f) >>= k\)

making it a composition of the continuation with the abs:

\(f >>= k \cdot \text{abs}\)

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

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

Move the abs operations inside the Lambdas

$(\lambda x \to f[x]) \cdot \text{abs}$

by changing the parameter of the lambda to have the abs applied.

$(\lambda x' \to \text{let } x=\text{abs } x' \text{ in } f[x])$

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

loopE (  
digitalReadE (rep button1) >>=  
  (\ a' \to digitalReadE (rep button2) >>=  
    (\ b' \to digitalWriteE (rep led) ((rep (abs a')) ||* (rep (abs b'))))) >>  
    delayMillisE (rep 1000))
Fuse Rep/Abs

Finally, with the abs moved into position, we are able to fuse the rep and the abs:

\[ rep(\text{abs } a) \] becomes \[ a \]

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

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

Conditionals are handled similarly to the primitive transformations:

\[
\text{forall (b :: \text{Bool}) (m1 :: \text{ExprB a => Arduino a})}
\quad (m2 :: \text{ExprB a => Arduino a}).
\]

\[
\text{if b then m1 else m2}
\quad =
\]

\[
\text{abs <$> ifThenElseE (rep b) (rep <$> m1)}
\quad (\text{rep <$> m2})
\]

\[
\text{forall (b :: \text{Bool}) (t :: \text{ExprB a => a})}
\quad (e :: \text{ExprB a => a}).
\]

\[
\text{if b then t else e}
\quad =
\]

\[
\text{abs$ ifB (rep b) (rep t) (rep e)}
\]
Recursion vs Iteration

• The Haskino EDSL includes an iteration primitive...

\[
\text{iterateE} :: \text{Expr} \ a \rightarrow \\
\quad (\text{Expr} \ a \rightarrow \text{Arduino} \ (\text{ExprEither} \ a \ b)) \rightarrow \\
\quad \text{Arduino} \ (\text{Expr} \ b)
\]

• However, we would like to write in a recursive style, as opposed to an iterative imperative style as follows:

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

blink :: Word8 \rightarrow \text{Arduino} ()
blink 0 = return ()
blink t = do
    digitalWrite led True
    delayMillis 1000
    digitalWrite led False
    delayMillis 1000
    blink $ t-1
```
Deep Recursion

```haskell
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))
```

```haskell
iterateE t $ do
  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)))
```
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
      _                 -> analogKey

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 ()))))))
  _                 -> analogKey
Mutual Recursion

- State 1
  - Any Key
  - Select
  - Right
  - Left

- State 2
  - Other Key

- State 3
  - Other Key

Power On
Mutual Recursion

```haskell
stateMachine :: LCD -> Arduino ()
stateMachine lcd = state1 $ keyValue KeyNone
  where
    state1 :: Word8 -> Arduino ()
    state1 k = do
      displayState lcd 1 k
      key <- analogKey
      case key of
        _ -> state2 key

state2 :: Word8 -> Arduino ()
state2 k = do
  displayState lcd 2 k
  key <- analogKey
  case key of
    1 -> state3 key
    5 -> state1 key
    _ -> state2 key

state3 :: Word8 -> Arduino ()
state3 k = do
  displayState lcd 3 k
  key <- analogKey
  case key of
    2 -> state2 key
    5 -> state1 key
    _ -> state3 key
```
Mutual Recursion

stateMachine_deep :: LCD -> Arduino (Expr ()
stateMachine_deep lcd = state1_deep (lit (keyValue KeyNone))
  where
    state1_deep :: Expr Word8 -> Arduino (Expr ()
    state1_deep k = state1_deep_mut (lit 0) k

    state2_deep :: Expr Word8 -> Arduino (Expr ()
    state2_deep k = state1_deep_mut (lit 1) k

    state3_deep :: Expr Word8 -> Arduino (Expr ()
    state3_deep k = state1_deep_mut (lit 2) k

    state1_deep_mut :: Expr Int -> Expr Word8 -> Arduino (Expr ()
    state1_deep_mut = iterateE i k state1_dep_mut_step
Mutual Recursion

\[
\text{state1\_deep\_mut\_step :: Expr Int -> Expr Word8 -> Arduino (ExprEither Word8 ())}
\]
\[
\text{state1\_deep\_mut\_step i k =}
\]
\[
\text{ifThenElseEither (i ==* (lit 0))}
\]
\[
(\text{transformed state 1 deep code})
\]
\[
\text{ifThenElseEither (i ==* (lit 1))}
\]
\[
(\text{transformed state 2 deep code})
\]
\[
(\text{transformed state 3 deep code})
\]
GHC Plugins

- GHC’s compiler plugin architecture allows the compiler user to modify or add passes to the compiler’s optimizer phase.

```haskell
type Plugin = [CommandLineOption] -> [Pass] -> CoreM [Pass]
```

- Each pass is a Core to Core transformation.

```haskell
type Pass = ModGuts -> CoreM ModGuts
```
Limitations

• Recursion Transformation only works on functions of zero or one arguments.
  • Addition of tuples to EDSL would remove limit.

• Three known untranslatable syntax constructs
  • `l ++ [c]` (ironically due to `build` construct)
  • Enum typeclass (limits on `fromEnum`)
  • `modifyRemoteRef` (translation of lambda function parameters)
  • These may be addressed by additions to the transformation logic/EDSL, and currently all have workarounds.
# Interpreter Resource Usage

## Flash Usage

<table>
<thead>
<tr>
<th></th>
<th>Shallow Haskino Interpreter in C</th>
<th>Shallow Haskino Interpreter in Haskino</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arduino Libraries</td>
<td>1032 bytes</td>
<td>1032 bytes</td>
</tr>
<tr>
<td>Haskino Runtime</td>
<td>-</td>
<td>3602 bytes</td>
</tr>
<tr>
<td>Applications</td>
<td>11396 bytes</td>
<td>18384 bytes (+61%)</td>
</tr>
<tr>
<td>Total Flash</td>
<td>12428 bytes</td>
<td>23018 bytes (+85%)</td>
</tr>
</tbody>
</table>

## Ram Usage

<table>
<thead>
<tr>
<th></th>
<th>Shallow Haskino Interpreter in C</th>
<th>Shallow Haskino Interpreter in Haskino</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>-</td>
<td>84 bytes</td>
</tr>
<tr>
<td>Message Buffers</td>
<td>32 bytes</td>
<td>96 bytes</td>
</tr>
<tr>
<td>Apps/Libs</td>
<td>502 bytes</td>
<td>561 bytes (+12%)</td>
</tr>
<tr>
<td>Total Static Ram</td>
<td>534 bytes</td>
<td>742 bytes (+39%)</td>
</tr>
<tr>
<td>Total Stack Ram</td>
<td>51 bytes</td>
<td>50 bytes</td>
</tr>
</tbody>
</table>
## Interpreter Performance

<table>
<thead>
<tr>
<th></th>
<th>Shallow Haskino Interpreter in C</th>
<th>Shallow Haskino Interpreter in Haskino</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing digitalRead</td>
<td>4.168 ms</td>
<td>4.093 ms</td>
<td>-1.8%</td>
</tr>
<tr>
<td>Communication Time</td>
<td>1.042 ms</td>
<td>1.042 ms</td>
<td></td>
</tr>
<tr>
<td>Host Time</td>
<td>0.133 ms</td>
<td>0.133 ms</td>
<td></td>
</tr>
<tr>
<td>Processing digitalWrite</td>
<td>8.204 ms</td>
<td>8.222 ms</td>
<td>+0.2%</td>
</tr>
<tr>
<td>Communication Time</td>
<td>6.163 ms</td>
<td>6.163 ms</td>
<td></td>
</tr>
<tr>
<td>Host Time</td>
<td>0.188 ms</td>
<td>0.188 ms</td>
<td></td>
</tr>
</tbody>
</table>
Code Sharing

- Some Deep Functions are “staged” by the plugin such that the Haskino Compiler is able to transform them into C functions as opposed to inlined code.

```
exampleFunc :: Expr Int -> Expr Int -> Arduino(Expr Int)
exampleFunc x y = return $ x + y
```

```
exampleFunc :: Expr Int -> Expr Int -> Arduino(Expr Int)
exampleFunc x y =
    app2Arg "exampleFunc" (exprArgType x) (exprArgType y)
    (exprRetType (exampleFunc_orig (remArg 0) (remArg 1)))
```

```
exampleFunc_orig :: Expr Int -> Expr Int -> Arduino(Expr Int)
exampleFunc_orig x y = return $ x + y
```
Flash Usage After Optimization

<table>
<thead>
<tr>
<th></th>
<th>Shallow Haskino Interpreter in C</th>
<th>Shallow Haskino Interpreter in Haskino</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arduino Libraries</strong></td>
<td>1032 bytes</td>
<td>1032 bytes</td>
</tr>
<tr>
<td><strong>Haskino Runtime</strong></td>
<td>-</td>
<td>3602 bytes</td>
</tr>
<tr>
<td><strong>Applications</strong></td>
<td>11396 bytes</td>
<td>12744 bytes</td>
</tr>
<tr>
<td><strong>Total Flash</strong></td>
<td>12428 bytes</td>
<td>17378 bytes</td>
</tr>
</tbody>
</table>
Future Work

• Implement Sharing Optimization

• Extend Translation to Higher Order Transversal functions.

• Generalization to non-monadic EDSLs
Publications

Accepted


Submitted

• M. Grebe, D. Young, and A. Gill, “Rewriting a shallow dsl using a ghc compiler extension,” extended version submitted to Computer Languages, Systems & Structures, Elsevier 2018
Conclusion

• One set of shallow source....

• Passed through a transformation plugin which is customizable for many EDSLs....

• Produces an language system with both ease of use, quick turnaround, and good performance.
Thank you for your attention

github.com/ku-fpg/haskino

http://ku-fpg.github.io/people/markgrebe/