Rewriting a Shallow DSL using a GHC compiler Extension

Mark Grebe, David Young, and Andy Gill
mark.grebe@ittc.ku.edu  david.young@ittc.ku.edu  andrew.gill@x.team

The University of Kansas
Haskino

- Haskino is an embedded domain specific language (EDSL), which provides a mechanism for programming the Arduino series of microcontrollers using monadic Haskell, instead of C.

- We use it as a test bed for our transformation techniques.
Haskino Overview

Haskell

- Shallow DSL
- GHC Core

GHC Core

- Shallow AST
- Remote Monad Send

Library

- Remote Monad Send

Compiler

- Trans-compiler

Arduino

- Firmware Interpreter
- C Code

Deep DSL

- Deep AST

Deep AST

- Plugin Translation

Shallow AST

- Plugin Translation

Library

- Remote Monad Send

- Trans-compiler

C Code

+ Runtime
Haskino Overview

- Haskell
- GHC Core
- Library
- Compiler
- Arduino

- Shallow DSL
- Shallow AST
- Remote Monad Send
- Firmware Interpreter

- Deep DSL
- Deep AST
- Trans-compiler
- C Code
- Runtime
Haskino Overview

Haskell

Shallow DSL

GHC Core

Shallow AST

Library

Remote Monad Send

Compiler

Arduino

Firmware Interpreter

Haskell

GHC

Core

Library

Compiler

Arduino

Shallow DSL

Deep DSL

Deep AST

Plugin Translation

Remote Monad Send

Compiler

Trans-compiler

C Code

Runtime

+
Haskino example

- To explain the shallow to deep transformation, we 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:

```haskell
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:

\[
\begin{align*}
\text{digitalWriteE} & :: \text{Expr Word8} \rightarrow \text{Expr Bool} \rightarrow \\
& \hspace{1cm} \text{Arduino (Expr ())} \\
\text{analogReadE} & :: \text{Expr Word8} \rightarrow \\
& \hspace{1cm} \text{Arduino (Expr Word16)}
\end{align*}
\]
We start with a basic example in the shallowly embedded 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 program runs in an intermediate representation back to the abstract type. This is illustrated in Figure 3.

Stating this in our DSL terms:

- 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 $\text{a}$ and $\text{Expr a}$
  - $\text{rep}$ is the equivalent of $\text{lit}$, and $\text{abs}$ corresponds to evaluation of the $\text{Expr}$. 

In the context of our expression language, that means that these conversions between the abstract and concrete types, and then outputs a value to a digital output. The output of the computations (in this example, the value of the button state in the Arduino). The shallow to deep transformation of a program written in a monadic EDSL uses a worker-wrapper based transformation.
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.

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

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

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

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

to ones of the form:

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

\[
\text{loop (}
\text{digitalRead button1 >>=}
(\a \rightarrow \text{digitalRead button2 >>=}
(\b \rightarrow \text{digitalWrite led (a || b))) >>
\text{delayMillis 100)})
\]

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

Translate the shallow operations to deep Expr operations:

\[
\text{rep } (x \ `\text{shallowOp}` y) \text{ transforms to } (\text{rep } x) \ `\text{deepOp}` (\text{rep } y)
\]

where the types of shallowOp and deepOp are:

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

\[
\begin{align*}
\text{loopE } ( & \quad \text{abs } \triangleleft\triangleright \text{digitalReadE } \text{(rep button1)} \triangleright\triangleright= \\
& \quad (\ \lambda \ a \to \text{abs } \triangleleft\triangleright \text{digitalReadE } \text{(rep button2)} \triangleright\triangleright= \\
& \quad \quad \lambda \ b \to \text{digitalWriteE } \text{(rep led)} \ (\text{rep } (a \ || \ b))) \triangleright\triangleright \ \\
& \quad \text{delayMillisE } \text{(rep 1000)})
\end{align*}
\]
Move Abs Through Binds

Move the abs operations through the monadic binds

\[(\text{abs } \langle\rangle f) \gg= k\]

making it a composition of the continuation with the abs:

\[f \gg= k \cdot \text{abs}\]

```
loopE (
    \text{abs } \langle\rangle \text{digitalReadE (rep button1)} \gg=
    (\ a \rightarrow \text{abs } \langle\rangle \text{digitalReadE (rep button2)} \gg=
    (\ b \rightarrow \text{digitalWriteE (rep led)} ((\text{rep a} | | \ast (\text{rep b}))) \gg
    \text{delayMillisE (rep 1000)})
)
```

```
loopE (
    \text{digitalReadE (rep button1)} \gg=
    (\ a \rightarrow \text{digitalReadE (rep button2)} \gg=
    (\ b \rightarrow \text{digitalWriteE (rep led)} ((\text{rep a} | | \ast (\text{rep b}))) . \text{abs}
    ) . \text{abs} \gg
    \text{delayMillisE (rep 1000)})
```
Move the abs inside the Lambdas

The lambdas may then be modified, changing the argument types to move the composed abs inside of the lambdas.

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

```
loopE (   
digitalReadE (rep button1) >>=   
  (\ a' -> digitalReadE (rep button2) >>=   
    (\ b' -> 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:

\[ rep (abs \ a) \] becomes \( \ a \)
Conditionals

Conditionals are handled similarly to the primitive transformations:

\[
\text{forall } (b :: \text{Bool}) (m1 :: \text{ExprB a} \Rightarrow \text{Arduino a}) \\
\quad (m2 :: \text{ExprB a} \Rightarrow \text{Arduino a}). \\
\text{if } b \text{ then } m1 \text{ else } m2 \\
\quad = \\
\text{abs } <$> \text{ifThenElseE} (\text{rep } b) (\text{rep } <$> m1) \\
\quad \quad (\text{rep } <$> m2)
\]

\[
\text{forall } (b :: \text{Bool}) (t :: \text{ExprB a} \Rightarrow \text{a}) \\
\quad (e :: \text{ExprB a} \Rightarrow \text{a}). \\
\text{if } b \text{ then } t \text{ else } e \\
\quad = \\
\text{abs } $ \text{ifB} (\text{rep } b) (\text{rep } t) (\text{rep } e)
\]
Recursion vs Iteration

• The Haskino EDSL includes an iteration primitive...

```haskell
iterateE :: Expr a ->
        (Expr a -> Arduino (Expr Either a b)) ->
        Arduino (Expr b)
```

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

```haskello
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
```
Recursion Transformation

```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
blinkE :: Expr Word8 -> Arduino (Expr ())
blinkE t =
  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))))
```
Shallow/Deep + Recursion Translation

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

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 ()))))
          (return (ExprLeft (lit ())))))
        (return (ExprLeft (lit ()))))
    (return (ExprLeft (lit ()))))})
GHC Plugin Passes

1. Simplifier Passes
2. Conditionals Pass
3. EDSL Primitives Pass
4. Return Translation Pass
5. Local Functions Pass
6. Rep Push and Abs Lambda Passes
7. Rep Abs Fusion Pass
8. Recursion Pass
Thank you for your attention

github.com/ku-fpg/haskino

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