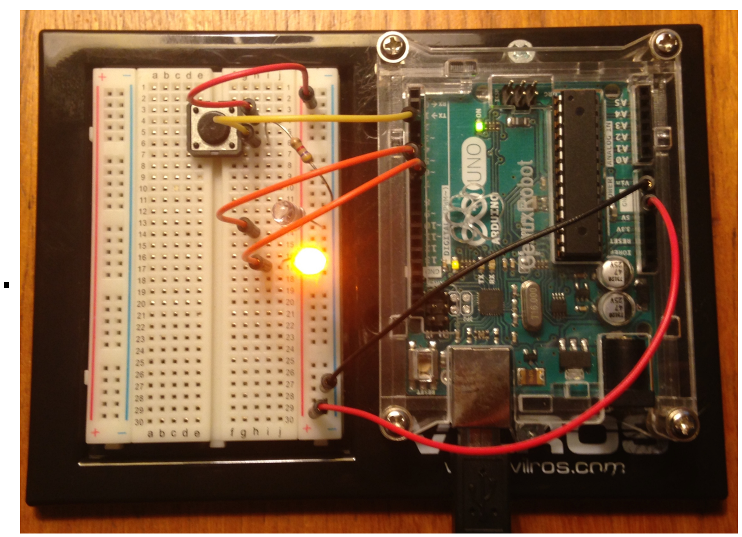
Domain Specific Languages for Small Embedded Systems

Mark Grebe
Department of Electrical Engineering and Computer Science

The University of Kansas
April 27, 2018

Small Embedded Systems

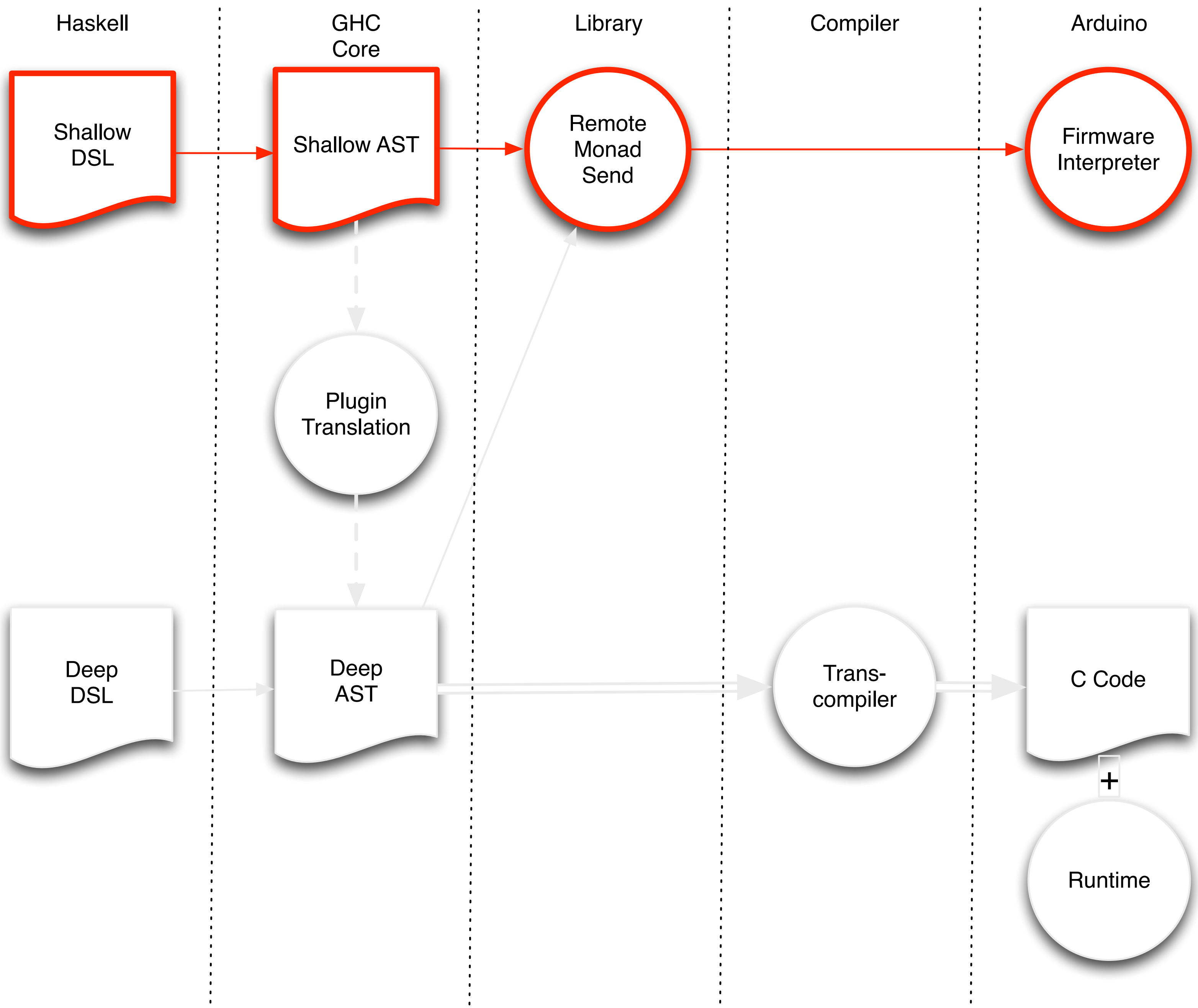
- Small, resource constrained embedded systems provide a challenge to programming with high level functional languages.
- Their small RAM and permanent storage resources make it impossible to run Haskell directly on them.
- Embedded Domain Specific Languages (EDSL) provides an alternative.
- Using an EDSL a user is able to write in a high level, functional host language.
- Execution can be through either interpretation or compilation.



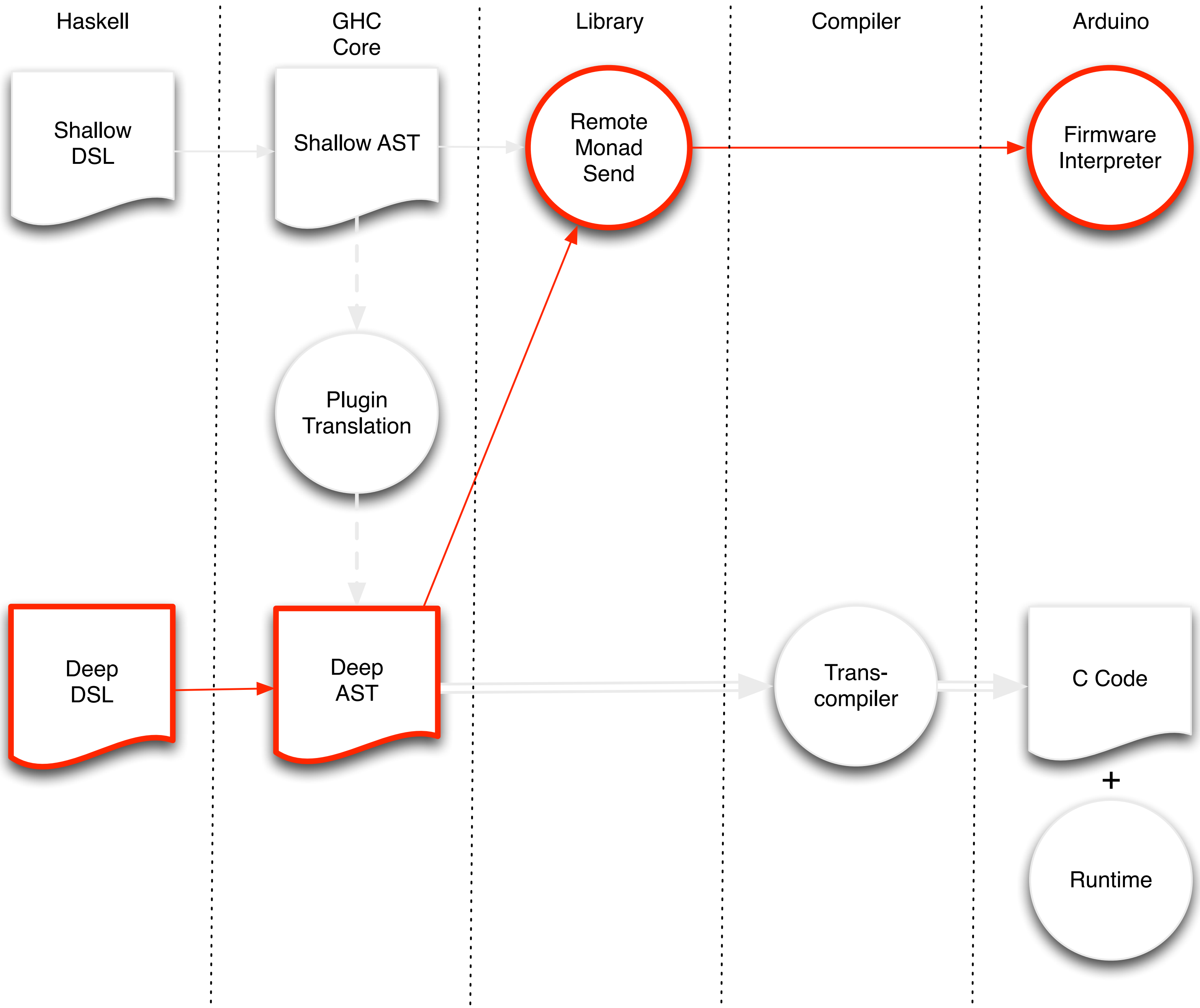
Embedded Domain Specific Languages

	Interpretation/Remote Execution	**Code Generation/ Compilation**
Shallow EDSL	*Examples* Blank Canvas hArduino Haxl *Advantages* Ease of development Quick turnaround	*Examples* Haskino *Advantages* Ease of development Performance Resource Optimization
Deep EDSL	*Advantages* Debugging	*Examples* Kansas Lava Feldspar Ivory *Advantages* Performance Resource Optimization

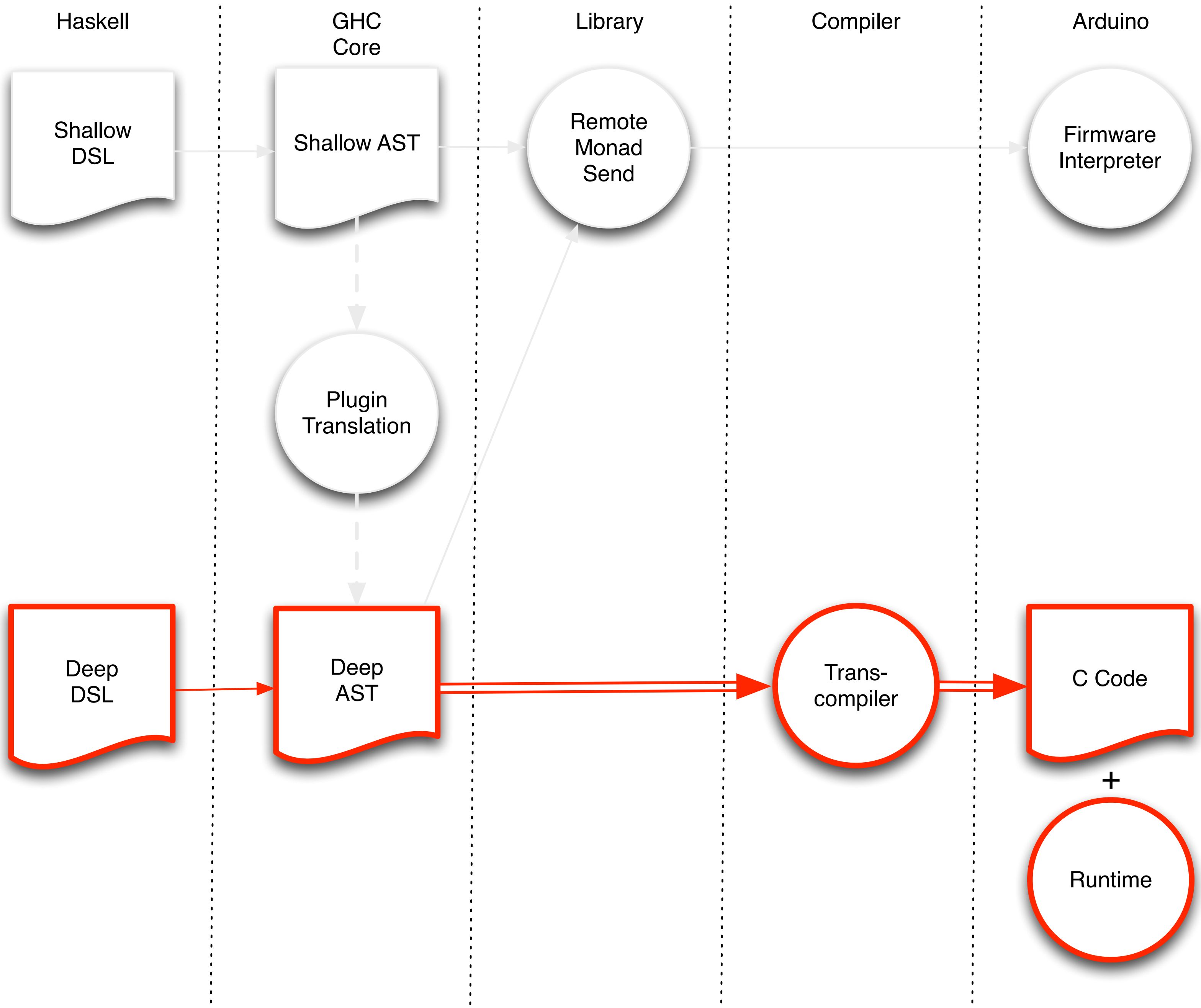
Haskino Overview



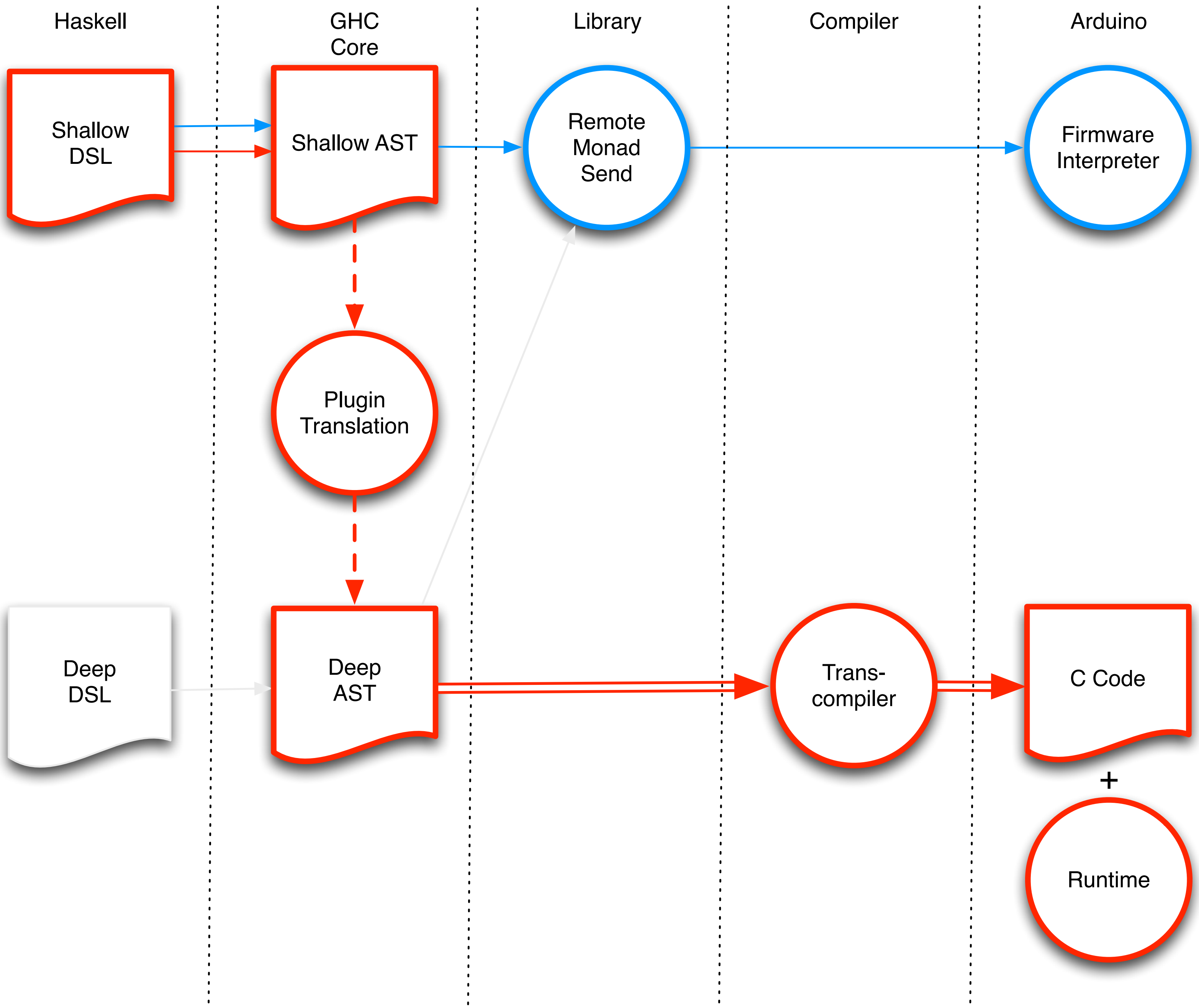
Haskino Overview



Haskino Overview



Haskino Overview



Remote Monads

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

```
digitalWrite :: Word8 -> Bool -> Arduino ()
send         :: ArduinoConnection -> Arduino a -> 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*

```
digitalRead :: Word8-> Arduino Bool
```

```
GHCi> send conn $ digitalRead 3
Arduino: Returns the state of Pin 3
```

Shallow Haskino example

- To 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:

```
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:

```
digitalWrite :: Word8 -> Bool -> Arduino ()
analogRead   :: Word8 -> Arduino Word16
```

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

```
digitalWriteE :: Expr Word8 -> Expr Bool ->
                 Arduino (Expr ())
analogReadE   :: Expr Word8 ->
                 Arduino (Expr Word16)
```

Expression Types

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

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

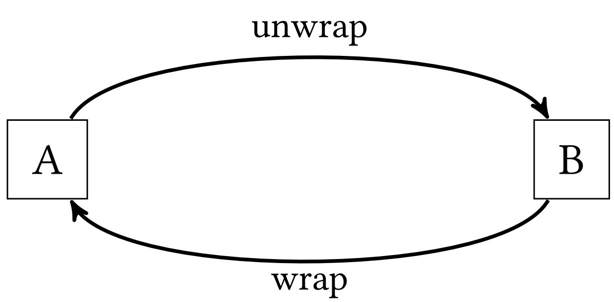
Conditionals become another data structure constructor when we move to the deep DSL:

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

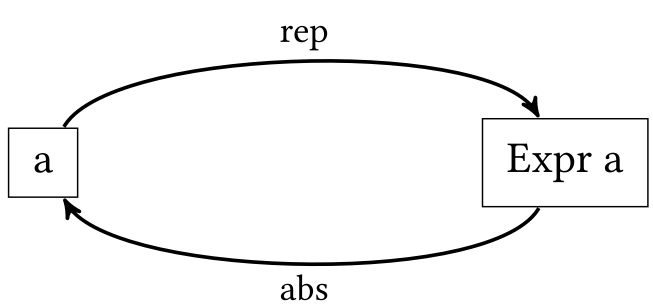
```
button <- digitalReadE (lit 2)
ifThenElseE button (digitalWriteE (lit 2) (lit True))
                   (digitalWriteE (lit 3) (lit True))
```

Transformations

Worker-Wrapper



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



- In our specific case, we move between *a* and *Expr a*
- *rep* is the equivalent of *lit,* and *abs* corresponds to evaluation of the 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.

```
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:

a1 -> ... -> an -> Arduino b

to ones of the form:

Expr a1 -> ... -> Expr an -> Arduino (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:

rep (x `shallowOp` y) transforms to *(rep x) `deepOp` (rep y)*

where the types of shallowOp and deepOp are:

shallowOp :: a -> b -> c and *deepOp :: Expr a -> Expr b -> Expr C*

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

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

Move Abs Through Binds

Move the abs operations through the monadic binds

(abs <$> f) >>= k

making it a composition of the continuation with the abs:

f >>= k . abs

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

```
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

(\ x -> f[x]) . abs

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

(\ x' -> let x=abs x' in f[x])

```
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*

```
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:

```
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)
```

```
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)
```

Recursion vs Iteration

- The Haskino EDSL includes an iteration primitive...

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

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

```
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
```

Deep Recursion

```
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))
```

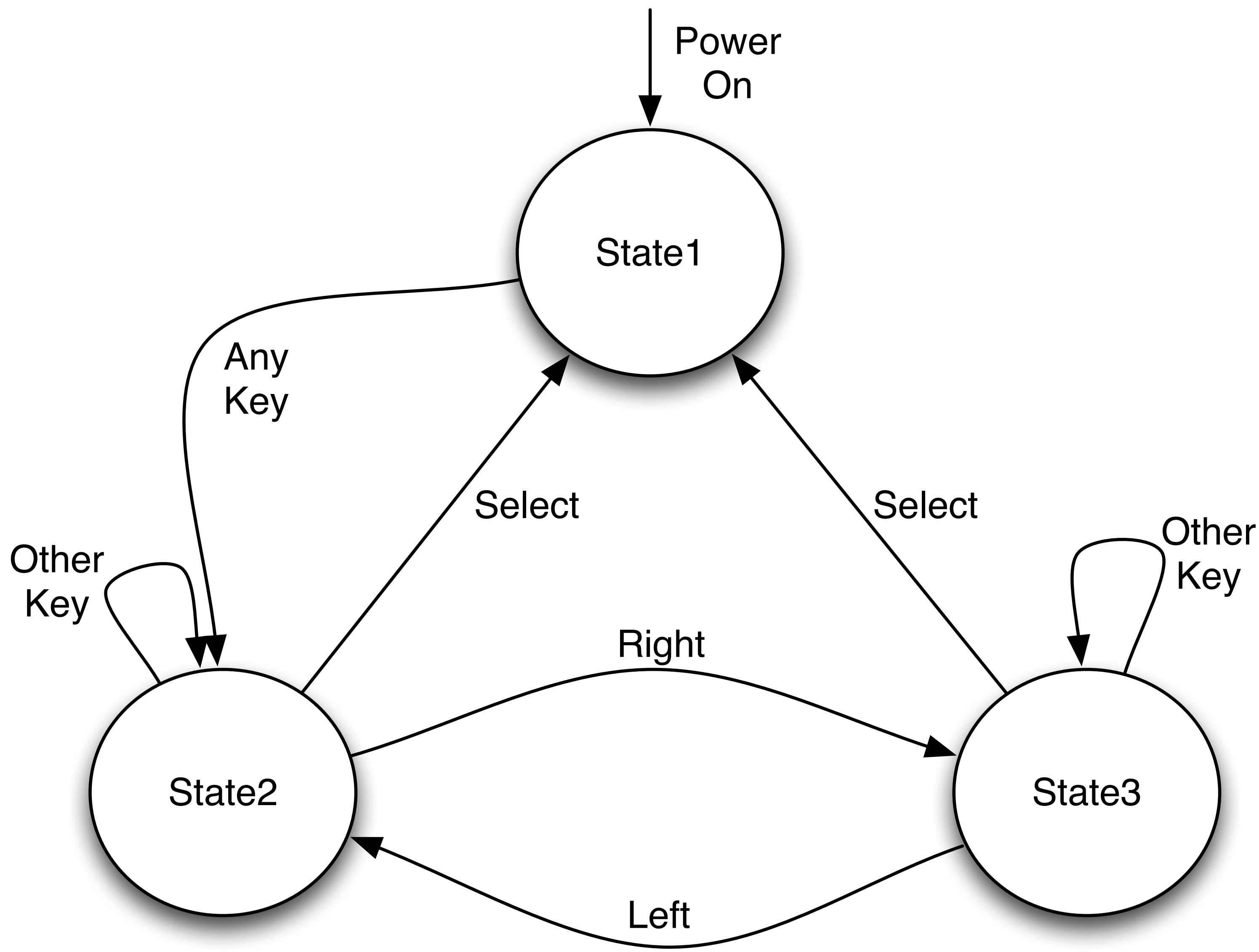
```
blinkE :: Expr Word8 -> Arduino (Expr ())
blinkE t =
  iterateE t $ do
    ifThenElseEither (t ==* rep 0)
      (return (ExprRight (rep ())))
      (do digitalWriteE (rep led) (rep True)
          delayMillisE (rep 1000)
          digitalWriteE (rep led) (rep False)
          delayMillisE (rep 1000)
          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 ())))))))
```

Mutual Recursion



Mutual Recursion

```
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 ke
```

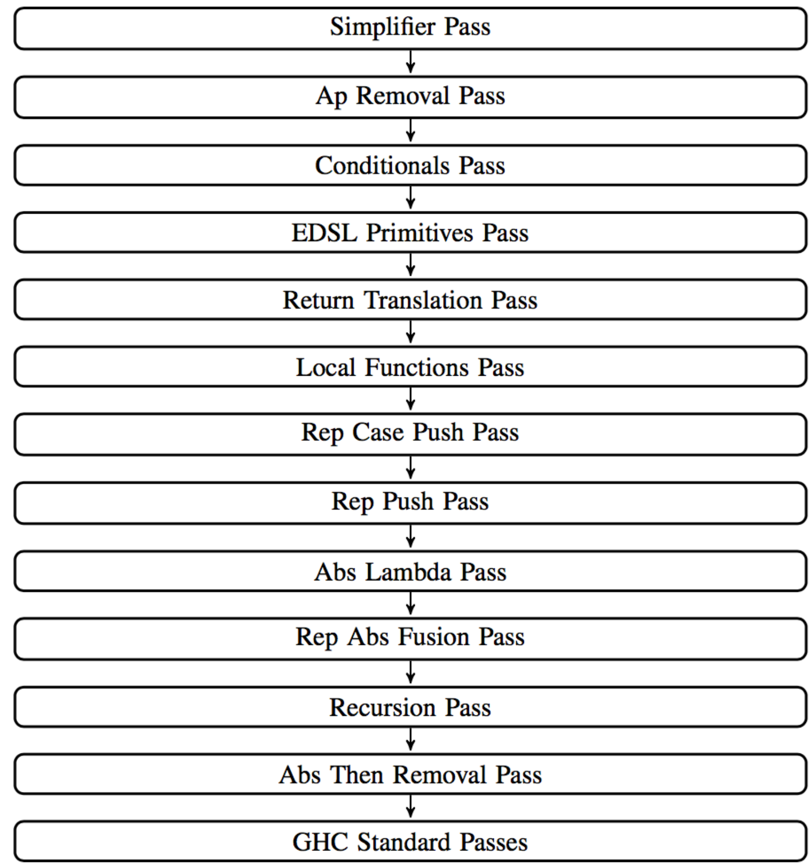
Mutual Recursion

```
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

```
state1_deep_mut_step :: Expr Int -> Expr Word8 -> Arduino (ExprEither Word8 ())
    state1_deep_mut_step i k =
        ifThenElseEither (i ==* (lit 0))
            (transformed state 1 deep code)
            (ifThenElseEither (i ==* (lit 1))
                (transformed state 2 deep code)
                (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.

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

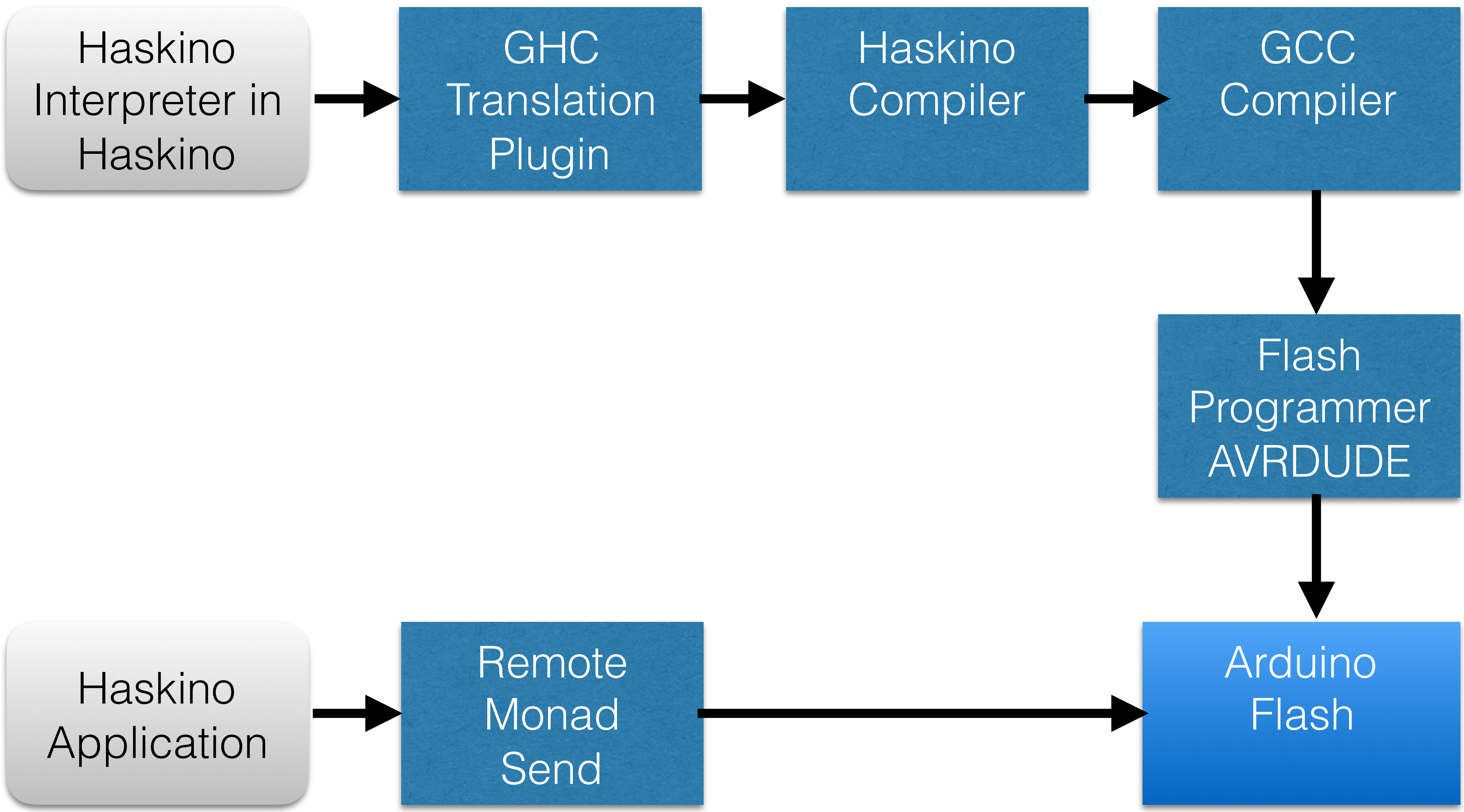
- Each pass is a Core to Core transformation.

```
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.

Haskino Bootstrap



Interpreter Resource Usage

Flash Usage

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino	
Arduino Libraries	1032 bytes	1032 bytes	
Haskino Runtime	-	3602 bytes	
Applications	11396 bytes	18384 bytes	**+61%**
Total Flash	12428 bytes	23018 bytes	+85%

Ram Usage

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino	
Scheduler	-	84 bytes	
Message Buffers	32 bytes	96 bytes	
Apps/Libs	502 bytes	561 bytes	**+12%**
Total Static Ram	534 bytes	742 bytes	+39%
Total Stack Ram	51 bytes	50 bytes	0%

Interpreter Performance

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino	
Processing digitalRead	4.168 ms	4.093 ms	**-1.8%**
Communication Time	1.042 ms	1.042 ms	
Host Time	0.133 ms	0.133 ms	
Processing digitalWrite	8.204 ms	8.222 ms	**+0.2%**
Communication Time	6.163 ms	6.163 ms	
Host Time	0.188 ms	0.188 ms	

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

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino	
Arduino Libraries	1032 bytes	1032 bytes	
Haskino Runtime	-	3602 bytes	
Applications	11396 bytes	12744 bytes	**+12%**
Total Flash	12428 bytes	17378 bytes	+40%

Future Work

- Implement Sharing Optimization
- Extend Translation to Higher Order Transversal functions.
- Generalization to non-monadic EDSLs

Publications

Accepted

- M. Grebe and A. Gill. Haskino: A Remote Monad for Programming the Arduino. In Practical Aspects of Declarative Languages, Springer (2016) 153-168

- M. Grebe and A. Gill. Threading the Arduino with Haskell. In Trends In Functional Programming, Springer 2017 (In Press)

- M. Grebe, D. Young, and A. Gill, “Rewriting a shallow dsl using a ghc compiler extension,” in Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, ser. GPCE 2017, New York, NY, USA: ACM, 2017, pp. 246–258.

- A. Gill, N. Sculthorpe, J. Dawson, A. Eskilson, A. Farmer, M. Grebe, J. Rosenbluth, R. Scott, J. Stanton. The remote monad design pattern. In Proceedings of the 8th ACM SIGPLAN Symposium on Haskel, pages 59–70. ACM, 2015.

- J. Dawson, M. Grebe, and A. Gill, “Composable network stacks and remote monads,” in Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, ser. Haskell 2017. New York, NY, USA: ACM, 2017, pp. 86–97.

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/