Rewriting a Shallow DSL using a GHC compiler Extension

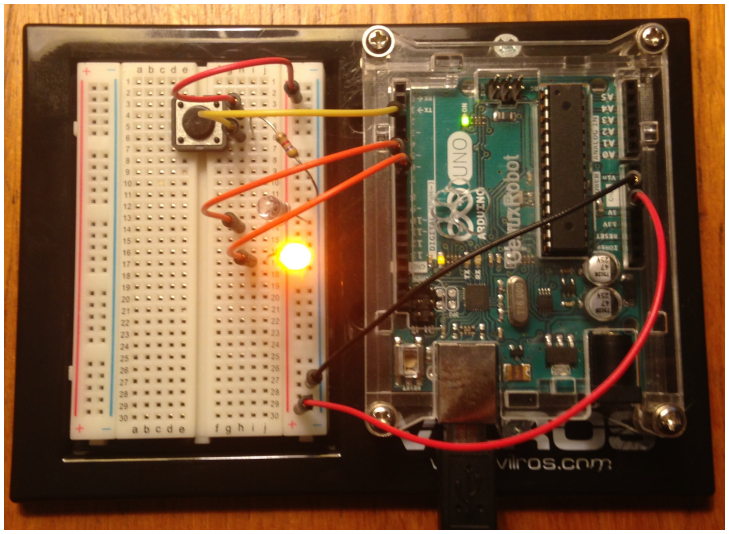
Mark Grebe, David Young, and Andy Gill
mark.grebe@ittc.ku.edu david.young@ittc.ku.edu andrew.gill@x.team

The University of Kansas

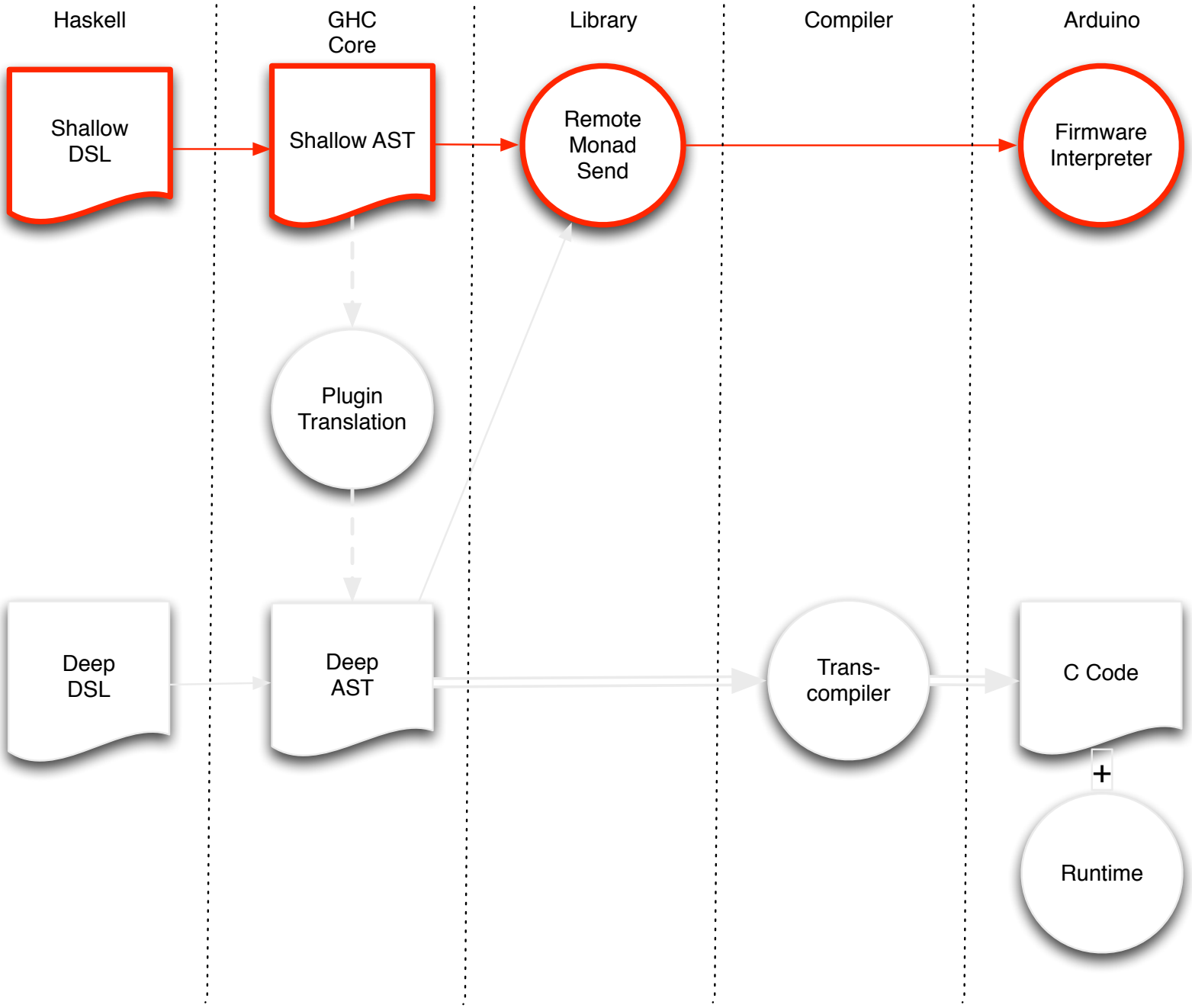
Haskino

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

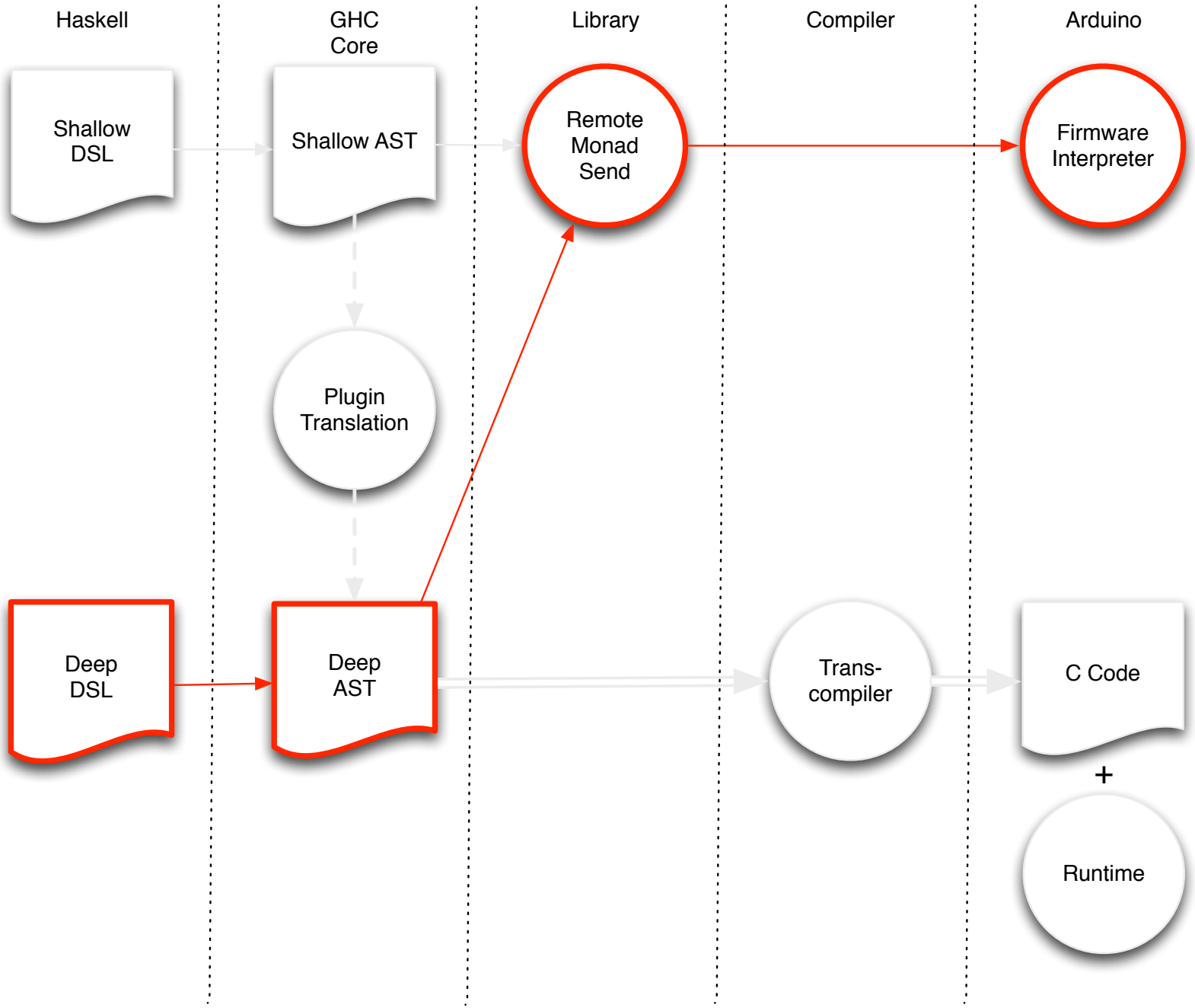
- We use it as a test bed for our transformation techniques.



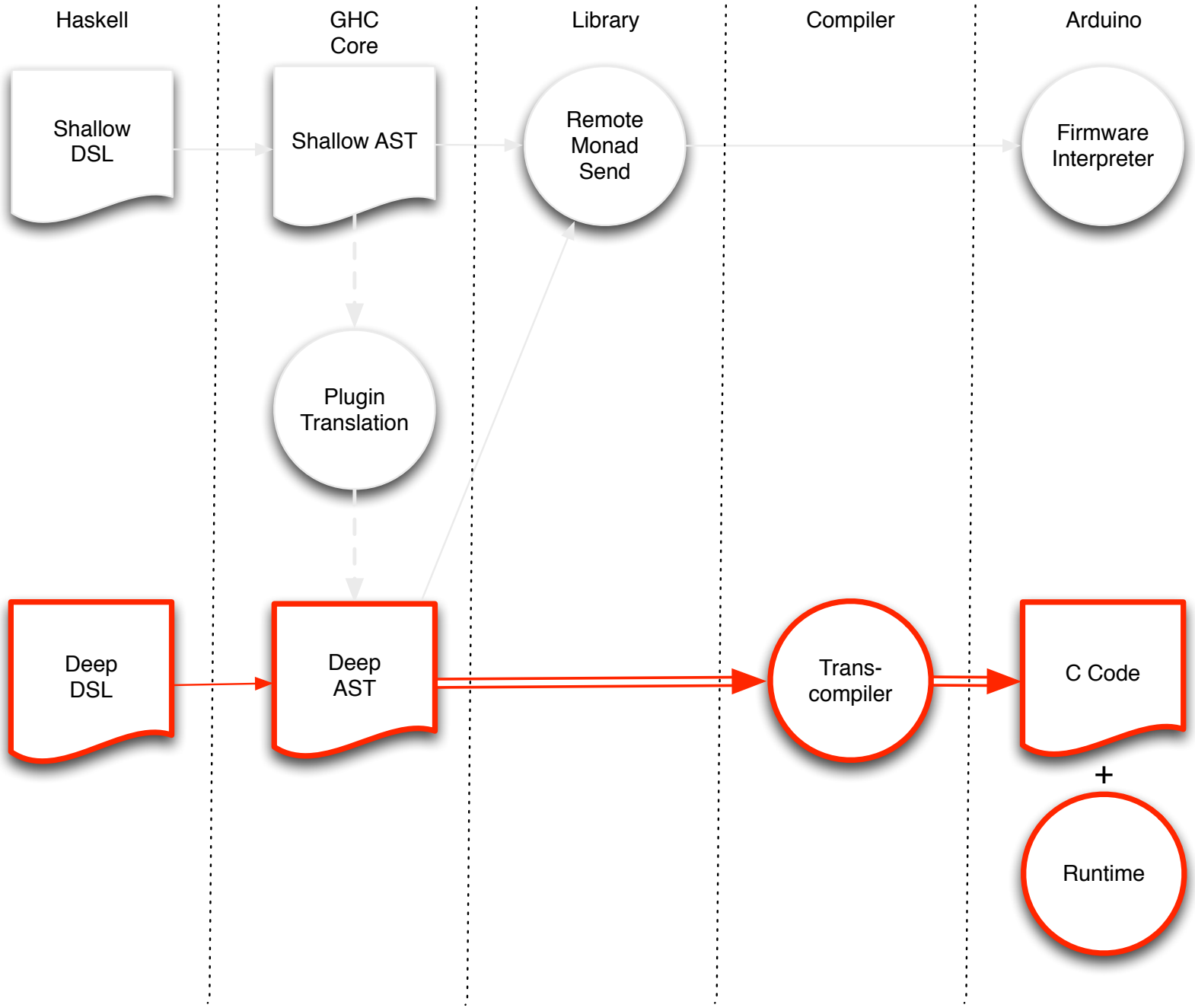
Haskino Overview



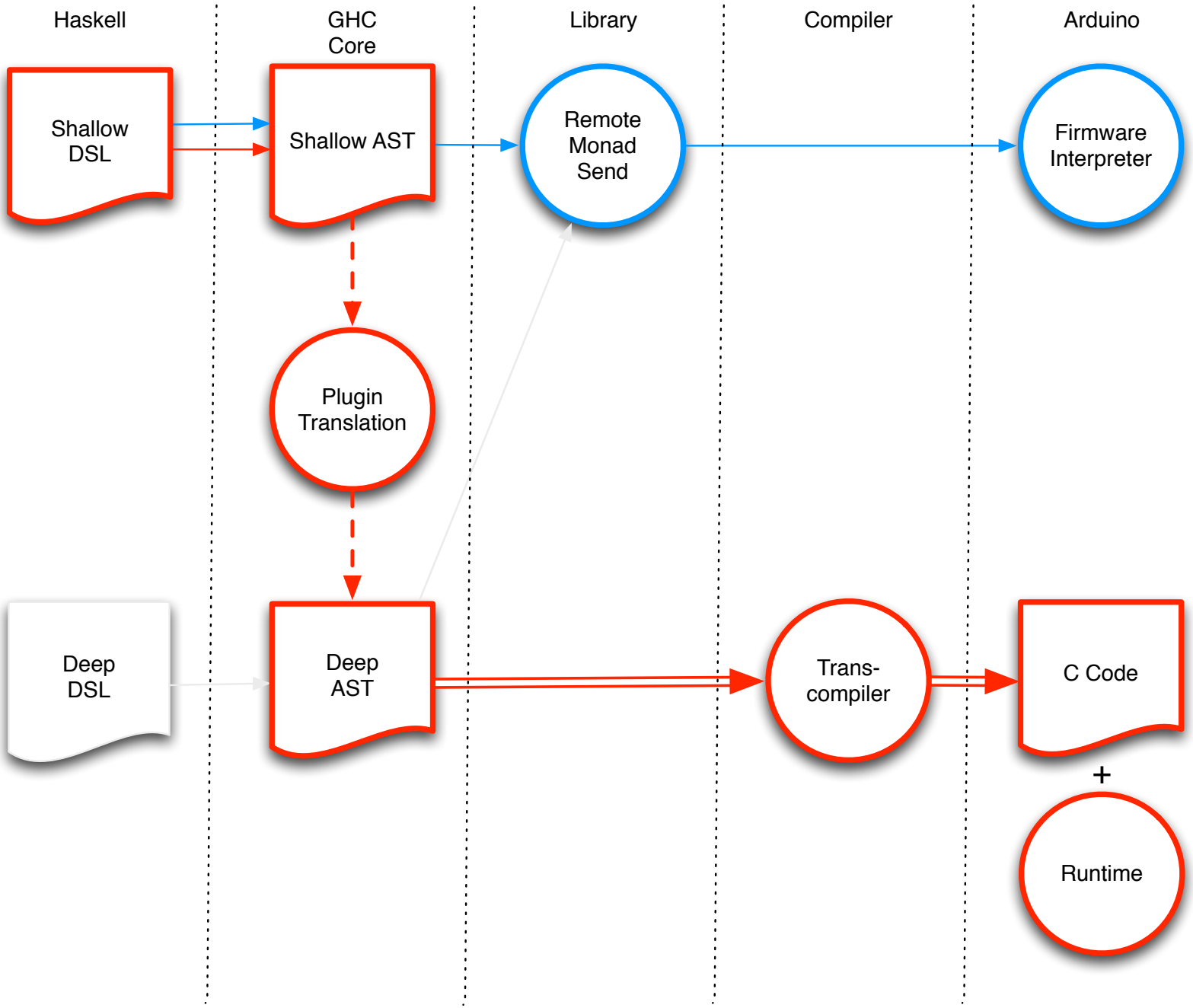
Haskino Overview



Haskino Overview



Haskino Overview



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:

```
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

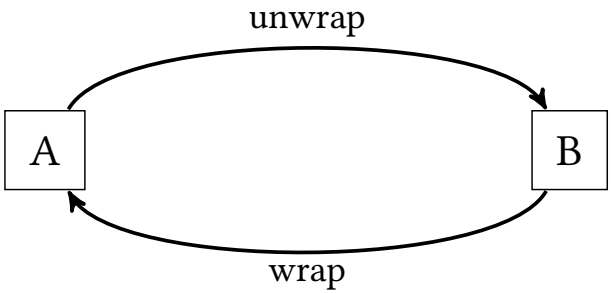
The tethered shallow Haskino uses commands and procedures such as:

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

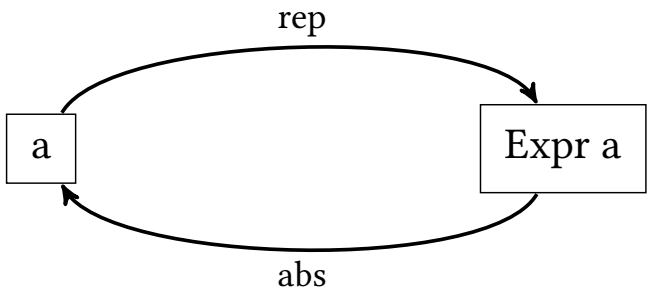
To move to the deeply embedded version, we instead use:

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

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

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*

```
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))
```

Conditionals

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)
```

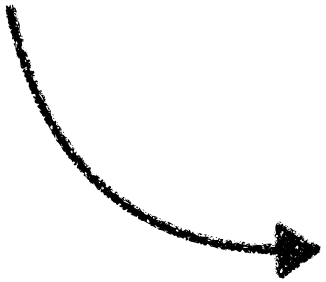
- However, we would like to write in a recursive style, as opposed to an iterative imperative style as follows:

```
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
```

Recursion Transformation

```
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

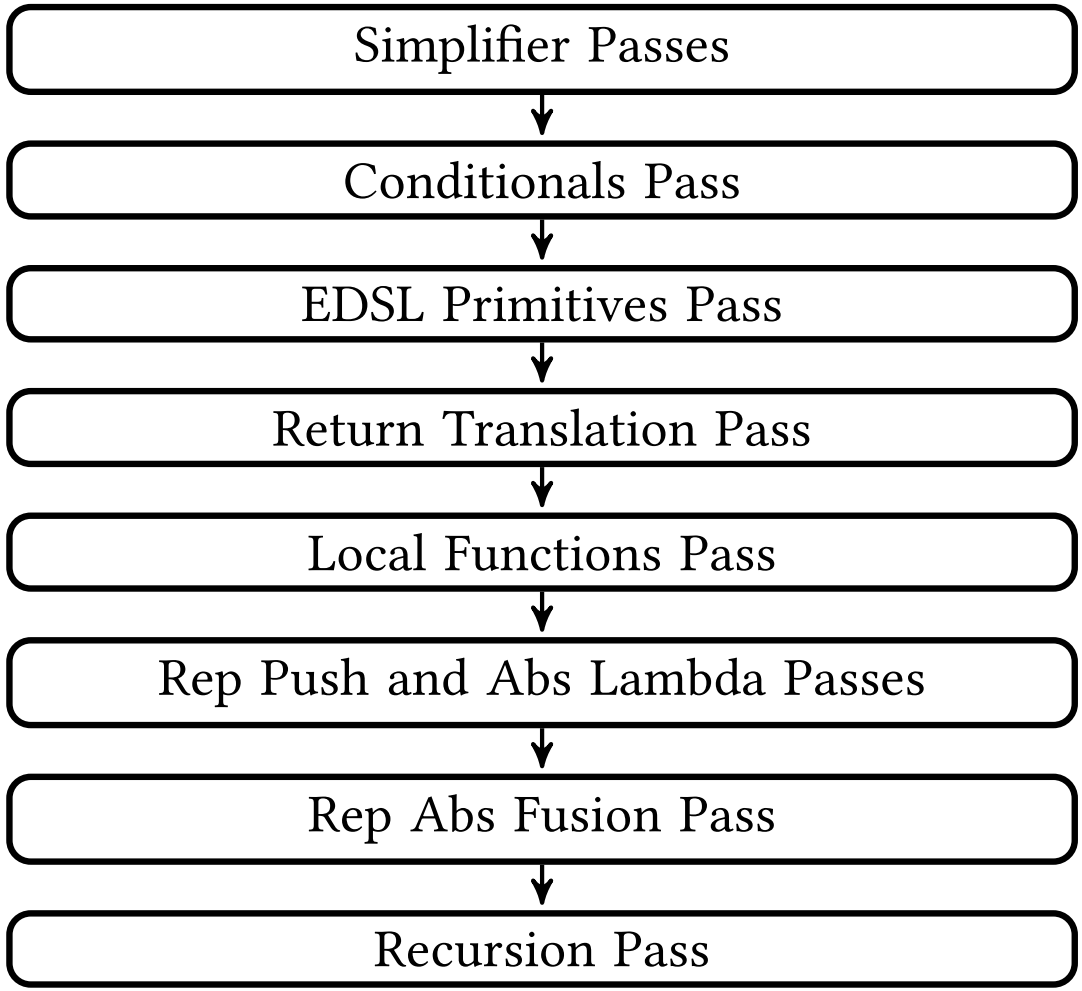
```
analogKey :: Arduino Word8
 analogKey = do
    v <- analogRead button2
    case v of
      _ | v < 30   -> return KeyRight
      _ | v < 150 -> return KeyUp
      _ | v < 350 -> return KeyDown
      _ | v < 535 -> return KeyLeft
      _ | v < 760 -> return KeySelect
      _           -> analogKey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

```
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 ()))))))))
```

GHC Plugin Passes



Thank you for your attention

github.com/ku-fpg/haskino

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