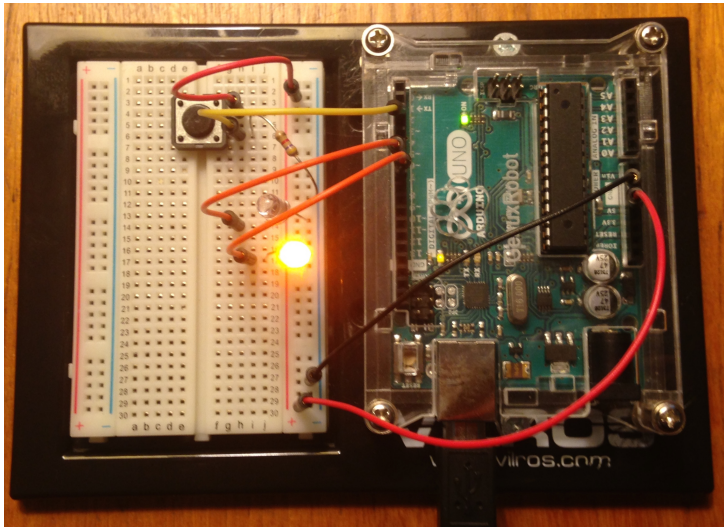
Threading the Arduino with Haskell

Mark Grebe and Andy Gill

The University of Kansas

Haskino

- Haskino provides a mechanism for programming the Arduino series of microcontrollers using monadic Haskell, instead of C.
- The current version provides two complementary methods:
  - An interpreter which uses an Arduino tethered to a host computer.
  - C Code generation which may then be compiled with a runtime and flashed for standalone operation.
- But first some background…



Haskino Single Threaded Example

```
exampleE :: IO ()
exampleE =
     withArduino False "/dev/cu.usbmodem1421" $ do
          let button = 2
          let led1 = 6
          let led2 = 7
          setPinModeE button INPUT
          setPinModeE led1 OUTPUT
          setPinModeE led2 OUTPUT
          loopE $ do
               ex <- digitalReadE button
               digitalWriteE led1 ex
               digitalWriteE led2 (notB ex)
               delayMillisE 100
```

Remote Monads

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

```
digitalWriteE :: Expr Word8 -> Expr Bool -> Arduino ()
send          :: ArduinoConnection -> Arduino a -> IO a
```

```
GHCi> send conn (digitalWriteE 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*

```
digitalReadE :: Expr Word8-> Arduino Bool
```

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

Remote Monads are about Bundling

At KU we have developed different strategies for bundling command and procedures.

*A **weak remote monad** is a remote monad that sends each of its remote calls individually to a remote interpreter*

*A **strong remote monad** is a remote monad that bundles all of its remote calls into packets of commands, punctuated by procedures, for remote execution.*

We are working on a third method of bundling, using an applicative bundling.

Haskell and Arduino Evolution

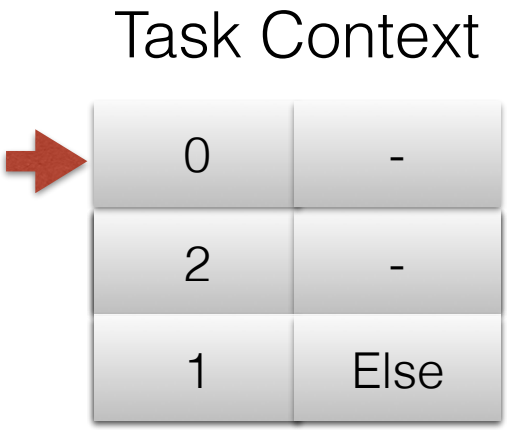
	Levent Erkök's hArduino	**Previous Haskino**	**Current Haskino**
Remote Monad Type	Weak	Strong	Strong
DSL Embedding	Shallow	Shallow/Deep	Deep
Firmware/ Protocol	Firmata	Haskino Interpreter	Haskino Interpreter/ Runtime
Interpreted/ Compiled	Interpreted	Interpreted	Interpreted/ Compiled
Threading	Single Threaded	Single Threaded	Multi-Threaded

Haskino Threads

- The previous version of Haskino inherited it's concept of threads from Firmata tasks.
- Tasks in Firmata are sequences of commands which can be executed at a future time, but they are only run to completion.
- We have subsequently extended Haskino to allow it to handle multiple threads of execution, with communication between the threads, and cooperative multitasking.
- The scheduler may be invoked, and rescheduling happen, as the result of a delay call or a semaphore procedure call

Interpreter Scheduling Saving Context

```
loopE $ do
  digitalWriteE led1 True
  av <- analogReadE inPin
  ifThenElse (av <* 100)
    (do digitalWriteE led2 True
     delayMillisE 1000)
    (do digitalWriteE led3 True
     delayMillisE 2000)
  digitalWriteE led1 False
  digitalWriteE led2 False
```



Interpreter Scheduling Restoring Context

```
loopE $ do
  digitalWriteE led1 True
  av <- analogReadE inPin
  ifThenElse (av <* 100)
    (do digitalWriteE led2 True
     delayMillisE 1000)
    (do digitalWriteE led3 True
     delayMillisE 2000)
  digitalWriteE led1 False
  digitalWriteE led2 False
```

Task Context

Inter-thread Communication

- Running multiple threads is of limited use if there is not a method of communication/synchronizing.
- Haskino provides a binary semaphore in both the interpreter and generated code.
- Using Haskino's remote references in conjunction with semaphores data may be passed between threads, and more complicated communications methods such as message queues constructed.
- Semaphores may also be used to communicate between a task and an Interrupt Service Routine, which may also be implemented as monadic tasks.

Inter-thread Communication

```
initExample :: Arduino ()
initExample = do
    let led = 13
    createTaskE 1 $ myTask1 led
    createTaskE 2 myTask2
    scheduleTaskE 1 1000
    scheduleTaskE 2 1050
```

```
myTask2 :: Arduino ()
myTask2 = do
    loopCount <- newRemoteRef $
                        lit (0 :: Word8)
    loopE $ do
        giveSemE semId
        t <- readRemoteRef loopCount
        writeRemoteRef loopCount $ t+1
        debugE $ showE t
        delayMillisE taskDelay
```

```
myTask1 :: Expr Word8 -> Expr Word32 ->
           Arduino ()
myTask1 led blinkDelay = do
    setPinModeE led OUTPUT
    i <- newRemoteRef $ lit (0 :: Word8)
    loopE $ do
        takeSemE semId
        writeRemoteRef i 0
        while i (\x -> x <* 3) (\x -> x + 1) $ do
            digitalWriteE led true
            delayMillisE blinkDelay
            digitalWriteE led false
            delayMillisE blinkDelay
```

Interpreter Limitations

- The interpreted version of the Haskino DSL provides a quick turnaround development environment.
- However, the interpreter takes most of the available flash memory space on the smaller Arduino boards.
- The only other memory available for program storage is EEPROM, which limits the size of programs.

Code Generation

- The limitations of the interpreter are overcome by using a compiler.
- Haskino provides a compiler that translates the same monadic code the interpreter uses into C code, which is then compiled and linked with a small Haskino runtime

```
compileProgram :: Arduino () -> FilePath -> IO ()

compile :: IO ()
compile = compileProgram initExample "semExample.ino"
```

Code Generation Initialization

- Setup initializes memory management, creates initial task and starts the scheduler

- Loop is unused in Haskino compiled sketches

```
void setup()
    {
    haskinoMemInit();
    createTask(255, haskinoMainTcb, HASKINOMAIN_STACK_SIZE,
               haskinoMain);
    scheduleTask(255, 0);
    startScheduler();
    }

void loop()
    {
    }
```

Code Generation
Main Task

- The Arduino () monad passed to the compile function is used to form the body of the main task

```
initExample :: Arduino ()
initExample = do
    let led = 13
    createTaskE 1 $ myTask1 led
    createTaskE 2 myTask2
    scheduleTaskE 1 1000
    scheduleTaskE 2 1050
```

```
void haskinoMain() {
    createTask(1, task1Tcb, TASK1_STACK_SIZE, task1);
    createTask(2, task2Tcb, TASK2_STACK_SIZE, task2);
    scheduleTask(1,1000);
    scheduleTask(2,1050);
    taskComplete();}
```

Code Generation Storage Allocation

- RemoteReference's are compiled into global C variables, named refX

```
myTask1 led = do
          ⋮
    i <- newRemoteRef $ lit (0::Word8)
```

```
uint8_t ref0;
void task1() {
          ⋮
    ref0 = 0;
```

- Binds are compiled into local variables, defined local to the code block in which they are used.

```
loopE $ do
          ⋮
    t <- readRemoteRef loopCount
    writeRemoteRef loopCount $ t+1
```

```
while (1)
    {
    uint8_t bind0;
          ⋮
    bind0 = ref0;
    ref0 = (bind0 + 1);
```

Code Generation
Task Control Block Allocation

- Compilation of the createTaskE procedure allocates a static task control block (TCB) as a global, which is passed to the runtime task creation routine.

- The TCB includes the task's stack, which is sized by the number of binds found in the task's monadic code.

```
createTaskE intTaskId intTask
```

```
void task1();
#define TASK1_STACK_SIZE 100
byte task1Tcb[sizeof(TCB) + TASK1_STACK_SIZE];

createTask(1, task1Tcb, TASK1_STACK_SIZE, task1);
```

Code Generation Scheduling/Runtime

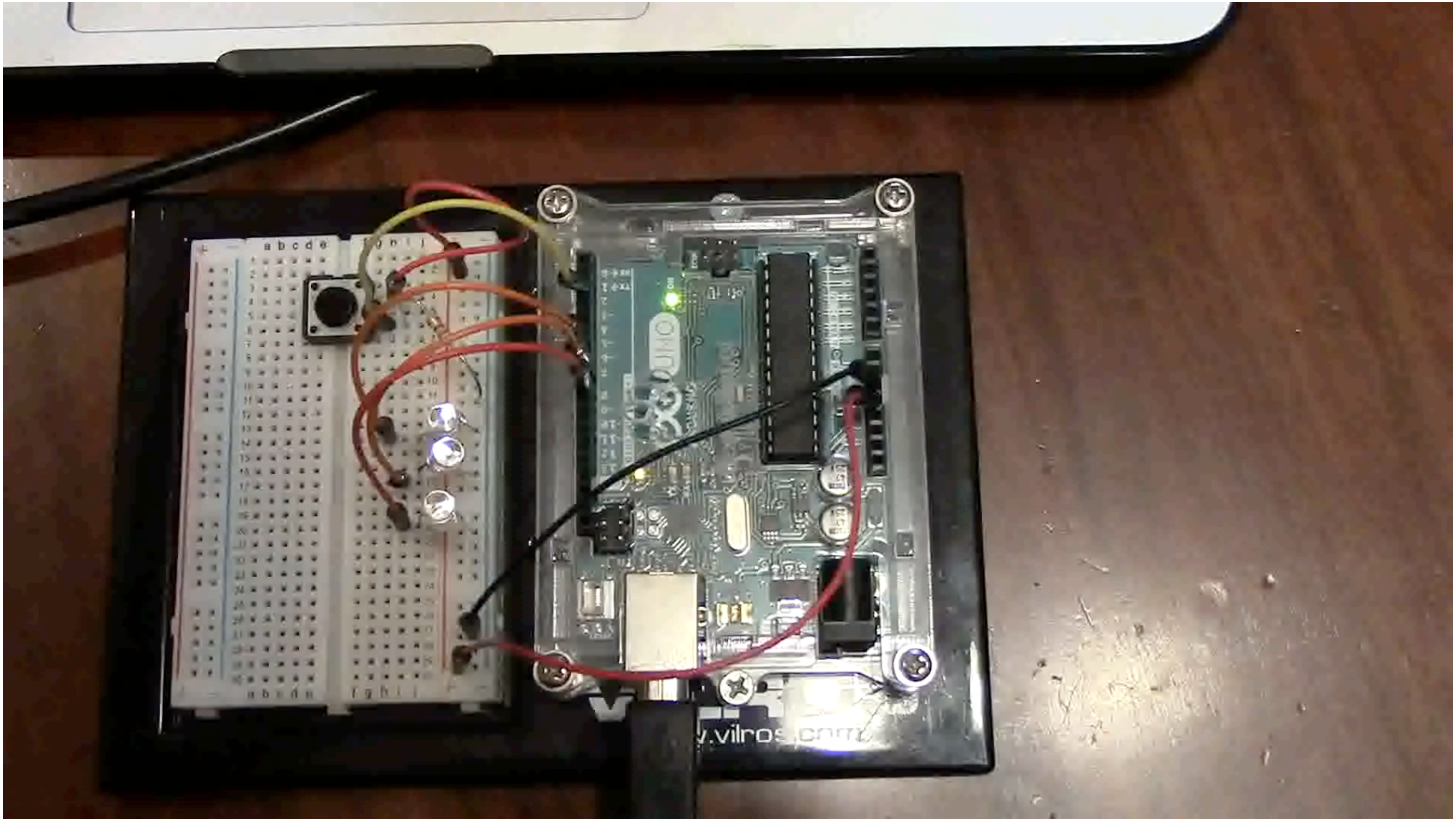
- The small Haskino runtime system used with the generated C code duplicates the scheduling capabilities of the Haskino interpreter
- This allow Haskino programs to be move seamlessly between the two environments.
- Like the Haskino interpreter, generated tasks are cooperative, only yielding the processor at delays and semaphore takes.

Multi-LED Example

```
ledTask :: Expr Word8 -> Expr Word32 -> Arduino ()
ledTask led delay = do
    setPinModeE led OUTPUT
    loopE $ do
        digitalWriteE led true
        delayMillisE delay
        digitalWriteE led false
        delayMillisE delay
```

```
initExample :: Arduino ()
initExample = do
    let led1 = 6
    let led2 = 7
    let led3 = 8
    createTaskE 1 $ ledTask led1  500 -- Create the tasks
    createTaskE 2 $ ledTask led2 1000
    createTaskE 3 $ ledTask led3 2000
    scheduleTaskE 1 1000 -- Schedule the tasks
    scheduleTaskE 2 2000
    scheduleTaskE 3 4000
```

Multi-LED Example



Conclusion

- The updated Haskino provides two complimentary methods of using Haskell as a development environment for Haskell software
  - An interpreter provides a method for quick prototyping in a tethered environment.
  - Compilation to intermediate C allows the programmer to bring the full power of Haskell to developing more complex standalone software for the Arduino
- Future work
  - We want to explore using HERMIT to semi-automatically translate from programs written in a more functional style, such as tail recursion instead of loops, to programs written using the deep embedding.
  - Extended scheduling to add thread priority and preemptive scheduling.

github.com/ku-fpg/haskino