Haskino: A Remote Monad for Programming the Arduino

Mark Grebe and Andy Gill

Information and Telecommunication Technology Center,
The University of Kansas, Lawrence, KS, USA,
first.last@ittc.ku.edu

Abstract. The Haskino library provides a mechanism for programming the Arduino microcontroller boards in high level, strongly typed Haskell instead of the low level C language normally used. Haskino builds on previous libraries for Haskell based Arduino programming by utilizing the recently developed remote monad design pattern. This paper presents the design and implementation of the two-level Haskino library. This first level of Haskino requires communication from the host running the Haskell program and the target Arduino over a serial link. We then investigate extending the initial version of the library with a deep embedding allowing us to cut the cable, and run the Arduino as an independent system.

Keywords: Haskell, Arduino, Remote Monad, Embedded Systems

1 Introduction

The Arduino line of microcontroller boards provide a versatile, low cost and popular platform for development of embedded control systems. Arduino boards have extremely limited resources that make running a high level functional language natively on the boards infeasible. Instead, the standard way of developing software for these systems is to use a C/C++ environment that is distributed with the boards. This paper documents our efforts to advance the use of Haskell to program the Arduino systems, starting with executing remote commands over a tethered serial port, towards supporting complete standalone systems.

To be specific, the most popular Arduino, the Arduino Uno, has a 16MHz clock rate, 2 KB of RAM, 32 KB of Flash, and 1 KB of EEPROM. This is cripplingly small by modern standards, but at a few dollars per unit and built-in A-to-D convertors and PWM support, many projects can be prototyped quickly and cheaply with careful programming. Using the Arduino itself as a testbed, we are interested in investigating how Haskell can contribute towards programming such small devices.

Programming the Arduino is, for the most part, straightforward imperative programming. There are side-effecting functions for reading and writing pins, supporting both analog voltages and digital logic. Furthermore, there are libraries for protocols like I^2C, and controlling peripherals, such as LCD displays.

We want to retain these APIs by providing an Arduino monad, which supports the low-level Arduino API, and allows programming in Haskell. Ideally, we want to cross-compile arbitrary Haskell code; the reality is we can get close using deeply embedded domain specific languages.

1.1 Outline

To make programming an Arduino accessible to functional programmers, we provide two complementary ways of programming a specific Arduino board.

- First, we provide a way of programming a *tethered* Arduino board, from directly inside Haskell, with the Arduino being a remote service. We start with the work of Levent Erkök, and his hArduino package [1], building on and generalizing the system by using a more efficient way of communicating, and generalizing the controls over the remote execution. We discuss this in Section 4.
- Second, we provide a way of out-sourcing entire groups of commands and control-flow idioms. This allows a user's Haskell program to program a board, then step back and let it run. It is this step – taming any allocation by using staging – that we want to be better able to understand, and later partially automate. We discuss this embedding in Section 6.

These two complimentarily ways provide a a gentler way of programming Arduinos, first using an API to prototype an idea, but with the full power of Haskell, then adjusting control flow and resource usage, to allow the exportation of the program. Both these methods use the remote monad design pattern [2] to provide the key capabilities.

In both systems, we build on the Firmata protocol and firmware [3], and provide a customizable interpreter that runs on the Arduino, written in C. In section 5 we discuss our runtime system, and compare it to previous works.

Our thesis is that structuring remote services in the manner outlined above allows for access to productive and powerful capabilities directly in Haskell, with a useable path to offshoring the entire remote computation. In section 8 we describe our most recent version of Haskino which extends the second API, and this is able to create a stored program on the Arduino which will run without being connected to the host.

2 Programming the Arduino in C

Programming the Arduino in C/C++ consists of defining two top level functions, `setup()`, which specifies the steps necessary to initialize the program, and `loop()`, which defines the main loop of the program. The Arduino environment provides a base set of APIs for controlling digital and analog input pins on the board, as well as standard libraries for other standard interfaces such as I^{2}C.

We present the following simple example of programming the Arduino in C/C++, and we will carry this example through the paper to demonstrate programming in several versions of our Haskino environment. This example has one

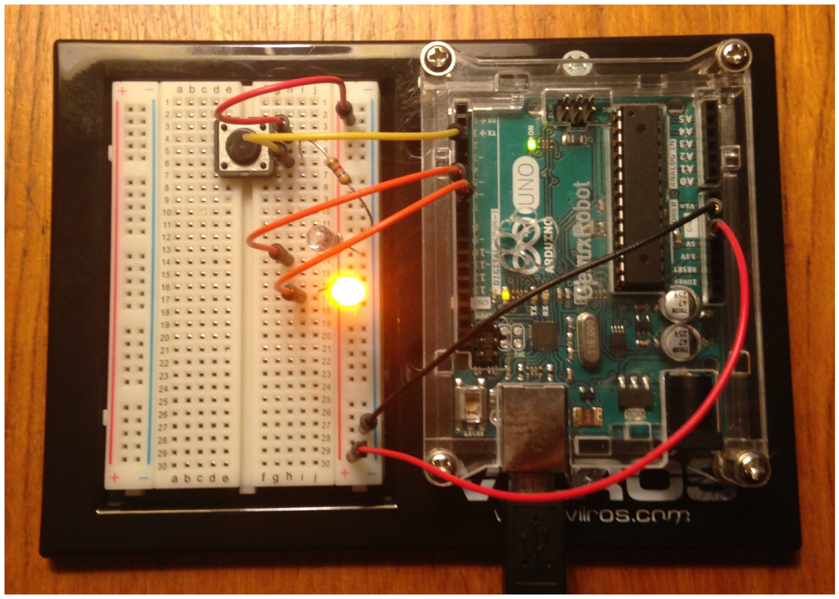


Fig. 1. Tethered Arduino Uni with Breadboard

digital input from a button, and two LED's for digital output. When the button is not pressed, LED1 will be off, and LED2 will be on. When the button is pressed, their states will be reversed. Figure 1 illustrates an Arduino Uno connected to two LEDs and a button running the example program. The constants 2, 6, and 7 in the program identify the numbers of the Arduino pins to which the button and LED's are connected.

```
int button = 2;
int led1 = 6;
int led2 = 7;

void setup() {
  pinMode(button, INPUT);
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
}

void loop() {
  int x;
  x = digitalRead(button);
  digitalWrite(led1, x);
  digitalWrite(led2, !x);
  delay(100);
}
```

3 The Remote Monad

A **remote monad**[2] is a monad that has its evaluation function in a remote location, outside the local runtime system. The key idea is to have a natural transformation, often called `send`, between *Remote* effect and *Local* effect.

$$\texttt{send} :: \forall\ a\ .\ Remote\ a \to Local\ a$$

The *Remote* monad encodes, via its primitives, the functionality of what can be done remotely, then the `send` command can be used to execute the remote commands. The `send` command is polymorphic, so it can be used to run individual commands, for their result, or to batch commands together. For example, Blank Canvas, our library for accessing HTML5 web-based graphics, uses the remote monad to provide a batchable remote service. Specifically, three representative functions from the API are:

```
send          :: Device -> Canvas a -> IO a
lineWidth     :: Double             -> Canvas ()
isPointInPath :: (Double,Double)    -> Canvas Bool
```

The `Canvas` is the remote monad, and there are three remote primitives given here, as well as bind and return. To use the remote monad, we use `send`:

```
send device $ do
   inside <- isPointInPath (0,0)
   lineWidth (if inside then 10 else 2)
```

The remote monad design pattern splits remote primitives into commands, where there is no interesting result value or temporal consequence, and procedures, which have a result value or temporal consequence. The design pattern then proposes different bundling strategies, based on the distinction between commands and procedures.

A *weak* remote monad is a remote monad that sends both commands and procedures one at a time, to be remotely executed. The design pattern is a way of structuring remote procedure calls, but has no performance advantage. A *strong* remote monad, however, bundles together chains of commands, terminated by an optional procedure, which has the interesting result. There are also other bundling strategies.

We have built a number of libraries using the remote monad design pattern. Blank Canvas is our Haskell library that provides the complete HTML5 Canvas API, using a strong remote monad that remotely calls JavaScript, and is fast enough to write small games. We have also built a general JSON-RPC framework in Haskell. In particular, the JSON-RPC protocol supports multiple batched calls, as well as individual calls, and our implementation uses monads and applicative functors to notate batching. We have also reimplemented the Minecraft API found in `mcpi`, adding a strong remote monad. We see many other other applications for the remote monad, beyond the topic of this paper, a remote Arduino monad.

4 The Arduino Remote Monad

The hArduino package, written by Levent Erkök, allows programmers to control Arduino boards through a serial connection. The serial protocol used between the host computer and the Arduino, and the firmware which runs on the Arduino, are together known as Firmata. Firmata was originally intended as a generic protocol for controlling microcontrollers from a host computer. It has become popular in the Arduino community, and programming interfaces for many programming languages have been developed for it. The hArduino library, using our terminology, uses a *weak* remote monad, and does not have a polymorphic send. Instead, once send is called, the function never terminates or returns values. This is our starting point.

Our first step in developing Haskino was to extend the hArduino library using the strong remote monad design pattern. The monad passed in hArduino represents the whole computation to be executed, which is then executed piecemeal by many individual remote calls. In contrast, Haskino's send function is able to send one or more commands terminated by a procedure which may return a value. This bundling of commands increases the efficiency of the communication, not requiring host interaction until a value is returned from the remote microcontroller.

With Haskino, to open a connection to an Arduino, `openArduino` is called passing a boolean flag for debugging mode, a file path to the serial port, and returns an ArduinoConnection data structure:

```
openArduino :: Bool -> FilePath -> IO ArduinoConnection
```

Once the connection is open, the `send` function may be called, passing an Arduino monad representing the computation to be performed remotely, and possibly returning a result.

```
send :: ArduinoConnection -> Arduino a -> IO a
```

The Arduino strong remote monad, like our other remote monad implementations, contains two types of monadic primitives, commands and procedures. An example of a command primitive is writing a digital value to a pin on the Arduino. In the strong version of Haskino, this has the following signature:

```
digitalWrite :: Pin -> Bool -> Arduino ()
```

The function takes the pin to write to and the boolean value to write, and returns a monad which returns unit. An example of a procedure primitive is reading the number of milliseconds since boot from the Arduino. The type signature of that procedure looks like:

```
millis :: Arduino Word32
```

Due to the nature of the Firmata protocol, the initial version of Haskino required a third type of monadic primitive. The Firmata protocol is not strictly a command and response protocol. Reads of analog and digital values from the Arduino are accomplished by issuing a command to start the reading process. Firmata will then send a message to the host at a set interval with the current requested value. In hArduino, and the initial version of Haskino, a background thread is used to read these returned values and store them in a local structure. To allow monadic computations to include reading of digital and analog values, the monadic primitive *local* is defined. A *local* is treated like a procedure from a bundling perspective, in that the send function sends any queued commands when the local is reached. However, unlike the procedure, the local is executed on the host, returning the digital or analog pin value that was stored by the background thread. Haskino also makes use of the these local type monadic primitives to provide a debug mechanism, allowing the language user to insert debug strings that will be printed when the they are reached during the send function processing.

The Arduino monad used in Haskino is defined using a GADT:

```
data Arduino :: * -> * where
    Command         :: Command                       -> Arduino ()
    Procedure       :: Procedure a                   -> Arduino a
    Local           :: Local a                       -> Arduino a
    Bind            :: Arduino a -> (a -> Arduino b) -> Arduino b
    Return          :: a                             -> Arduino a

instance Monad Arduino where
        return = Return
        (>>=) = Bind
```

The instance definition for the Monad type class is shown above, but similar definitions are also defined for the Applicative, Functor, and Monoid type classes as well. Each of the types of monadic primitives described earlier in this section is encoded as a sub data type, Command, Procedure, and Local. The data types for Command, Procedure and Local are shown below, with only a subset of their actual constructors as examples.

```
data Command =
     DigitalWrite Pin Bool
     | AnalogWrite Pin Word16

data Procedure :: * -> * where
     Millis       :: Procedure Word32
     | Micros     :: Procedure Word32

data Local :: * -> * where
     DigitalRead      :: Pin -> Local Bool
     | AnalogRead     :: Pin -> Local Word16
```

Finally, the API functions which are exposed to the programmer are defined in terms of these constructors, as shown for the example of `digitalWrite` below:

```
digitalWrite :: Pin -> Bool -> Arduino ()
digitalWrite p b = Command $ DigitalWrite p b
```

hArduino used the original version of Firmata, known as Standard Firmata. The initial version of Haskino used a newer version of Firmata, called Configurable Firmata, adding the ability to control additional Arduino libraries, such as I^2C, OneWire and others. In addition to providing control of new interfaces, it introduces a basic scheduling system. Firmata commands are able to be combined into tasks, which then can be executed at a specified time in the future. Haskino makes use of this capability to specify a monadic computation which is run at a future time. The strong version is limited in what it can do with the capability, as it has no concept of storing results of computations on the remote system for later use, or of conditionals. However, we will return to this basic scheduling capability in Section 5 and Section 6, when we describe enhancements that are made in our deep version of Haskino.

To demonstrate the use of Haskino, we return to the simple example presented in Section 2, this time written in the strong version of the Haskino language.

```
example :: IO ()
example = withArduino False "/dev/cu.usbmodem1421" $ do
          let button = 2
          let led1 = 6
          let led2 = 7
          setPinMode button INPUT
          setPinMode led1 OUTPUT
          setPinMode led2 OUTPUT
          loop $ do
               x <- digitalRead button
               digitalWrite led1 x
               digitalWrite led2 (not x)
               delayMillis 100
```

This example uses the Haskino convenience function, `withArduino`, which calls `openArduino` , and then calls `send` with the passed monad:

```
withArduino :: Bool -> FilePath -> Arduino () -> IO ()
```

The `setPinMode` commands configure the Arduino pins for the proper mode, and will be sent as one sequence by the underlying send function. The `loop` primitive is similar to the forever construct in Control.Monad, and executes the sequence of commands and procedures following it indefinitely. The `digitalRead` function is a procedure, so it will be sent individually by the `send` function. The two `digitalWrite` commands following the `digitalRead` will be bundled with the `delayMillis` procedure.

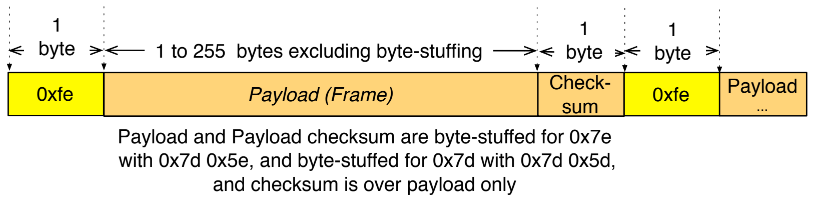

1 byte
1 to 255 bytes excluding byte-stuffing
1 byte
1 byte
0xfe
Payload (Frame)
Check-sum
0xfe
Payload ...
Payload and Payload checksum are byte-stuffed for 0x7e with 0x7d 0x5e, and byte-stuffed for 0x7d with 0x7d 0x5d, and checksum is over payload only

Fig. 2. Haskino Framing

5 Haskino Firmware and Protocol

We want to move from sending bundles of commands to our Arduino, to sending entire control-flow idioms, even whole programs, as large bundles. We do this by using deep embedding technology, embedding both a small expressing language, and deeper Arduino primitives.

Specifically, to move Haskino from a straightforward use of the strong remote monad to a deeper embedding, required extending the protocol used for communication with the Arduino to handle expressions and conditionals. The Firmata protocol, while somewhat expandable, would have required extensive changes to accommodate expressions. Also, since it was developed to be compatible with MIDI, it uses a 7 bit encoding which added complexity to the implementation on both the host and Arduino sides of the protocol. As we had no requirement to maintain MIDI compatibility, we determined that it would be easier to develop our own protocol specifically for Haskino.

Like Firmata, the Haskino protocol sends frames of data between the host and Arduino. Commands are sent to the Arduino from the host, with no response expected. Procedures are sent to the Arduino as a frame, and then the host waits for a frame from the Arduino in reply to indicated completion, returning the value from procedure computation.

Instead of 7 bit encoding, the frames are encoded with an HDLC (High-level Data Link Control) type framing mechanism. Frames are separated by a hex 0x7E frame flag. If a 0x7E appears in the frame data itself, it is replaced by an escape character (0x7D) followed by a 0x5E. If the escape character appears in the frame data, it is replaced by a 0x7D 0x5D sequence. The last byte of the frame before the frame flag is a checksum byte. Currently, this checksum is an additive checksum, since the error rate on the USB based serial connection is relatively low, and the cost of a CRC computation on the resource limited Arduino is relatively high. However, for a noisier, higher error rate environment, a CRC could easily replace the additive checksum. Figure 2 illustrates the framing structure used.

The new Haskino protocol also makes another departure from the Firmata style of handling procedures which input data from the Arduino. With the deep embedded language being developed, results of one computation may be used in another computation on the remote Arduino. Therefore, the continuous, periodic style of receiving digital and analog input data used by Firmata does not make sense for our application. Instead, digital and analog inputs are requested each time they are required for a computation. Although, this increases the communication overhead for the strong remote monad implementation, it enables the deep implementation, and allows a common protocol to be used by both.

The final design decision required for the protocol was to determine if the frame size should have a maximum limit. As the memory resources on the Arduino are limited, the frame size of the protocol a maximum frame size of 256 bytes was chosen to minimize the amount of RAM required to store a partially received frame on the Arduino.

The basic scheduling concept of Firmata was retained in the new protocol as well. The CreateTask command creates a task structure of a specific size. The AddToTask command adds monadic commands and procedures to a task. Multiple AddToTask commands may be used for a task, such that the task size is not limited by the maximum packet size, but only by the amount of free memory on the Arduino. The ScheduleTask command specifies the future time offset to start running a task. Multiple tasks may be defined, and they run till completion, or until they delay. A delay as the last action in a task causes it to restart. Commands and procedures within a task message use the same format as the command sent in a individual frame, however, the command is proceeded by a byte which specifies the length of the command.

The new protocol was implemented in both Arduino firmware and the strong remote monad version of the Haskell host software, producing the second version of Haskino.

6 Deep EDSL

To move towards our end goal of writing an Arduino program in Haskell that may be run on the Arduino without the need of a host computer and serial interface, we needed to move from the strong remote monad used in the first two versions of the library. A deep embedding of the Haskino language allows us to deal not just with literal values, but with complex expressions, and to define bindings that are used to retain results of computations remotely.

To accomplish this goal, we have extended the command and procedure monadic primitives to take expressions as parameters, as opposed to simple values. For example, the `digitalWrite` command described earlier now becomes the `digitalWriteE` command:

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

Procedure primitives now also return Expr values, so the `millis` procedure described earlier now becomes the `millisE` procedure defined as:

```
millisE :: Arduino (Expr Word32)
```

The Expr data type is used to express arithmetic and logical operations on both literal values of a data type, as well as results of remote computations of the same data type. Expr is currently defined over boolean and unsigned integers of length 8, 16 and 32, as these are the types used by the builtin Arduino API. It could be easily extended to handle signed types as well. For booleans, the standard logical operations of not, and, and or are defined. Integer operations include addition, subtraction, multiplication and division, standard bitwise operations, and comparison operators which return a boolean. Type classes and type families are defined using the Data.Boolean [4] package such that operations used in expressions may be written in same manner that operations on similar data types are written in native Haskell. For example, the following defines two expressions of type Word8, and then defines a boolean expression which determines if the first expression is less than the second.

```
a :: Expr Word8
a = 4 + 5 * 9

a :: Expr Word8
b = 6 * 7

c :: Expr Bool
c = a <* b
```

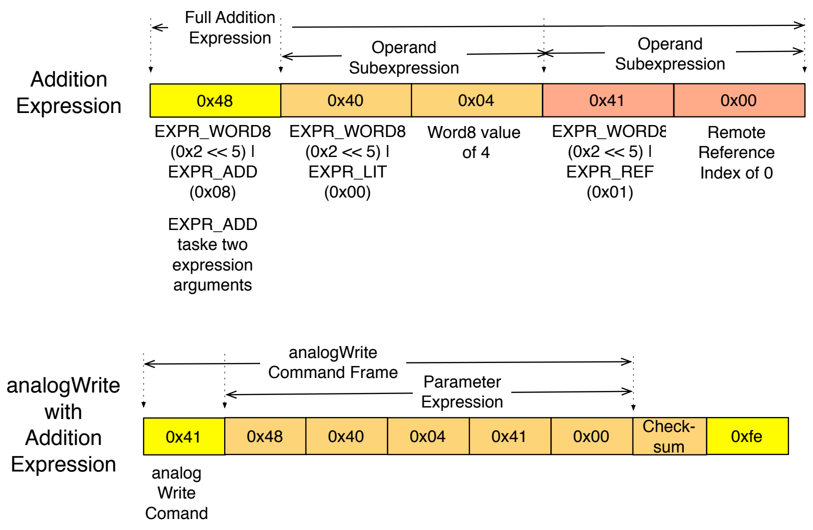
Strong remote monad commands may be defined in terms of their deep counterparts, allowing both to coexist in the deep embedded version. For example:

```
digitalWrite :: Word8 -> Bool -> Arduino ()
digitalWrite p b = digitalWriteE (lit p) (lit b)
```

The second component of the deep embedding is the ability to define remote bindings which allow us to use the results of one remote computation in another. For this, we define a RemoteReference typeclass, with an API that is similar to Haskell's IORef. With this API, remote references may be created and easily read and written to.

```
class RemoteReference a where
    newRemoteRef    :: Expr a -> Arduino (RemoteRef a)
    readRemoteRef   :: RemoteRef a -> Arduino (Expr a)
    writeRemoteRef  :: RemoteRef a -> Expr a -> Arduino ()
    modifyRemoteRef :: RemoteRef a -> (Expr a -> Expr a) ->
                       Arduino ()
```

The final component of the deep embedding is adding conditionals to the language. Haskino defines three types of conditional monadic structures, an If-Then-Else structure, and a While structure, and a LoopE structure. The `while` structure emulates while loops, and it takes a RemoteRef, a function returning a boolean expression to determine if the loop terminates, a function which updates



Addition Expression
Full Addition Expression
Operand Subexpression
Operand Subexpression
0x48
0x40
0x04
0x41
0x00
EXPR_WORD8 (0x2 << 5) I EXPR_ADD (0x08)
EXPR_WORD8 (0x2 << 5) I EXPR_LIT (0x00)
Word8 value of 4
EXPR_WORD8 (0x2 << 5) I EXPR_REF (0x01)
Remote Reference Index of 0
EXPR_ADD taske two expression arguments
analogWrite with Addition Expression
analogWrite Command Frame
Parameter Expression
0x41
0x48
0x40
0x04
0x41
0x00
Check-sum
0xfe
analog Write Comand

Fig. 3. Example of Expression Encoding

the remote reference at the end of each loop, and a Arduino () monad which specifies the loop body. The `loopE` structure provides a deep analog of the `loop` structure used in the strong remote monad version.

```
ifThenElse :: Expr Bool -> Arduino () -> Arduino () -> Arduino ()
while :: RemoteRef a -> (Expr a -> Expr Bool) ->
         (Expr a -> Expr a) -> Arduino () -> Arduino ()
loopE :: Arduino () -> Arudino ()
```

Changes to the Haskino protocol and firmware were also required to implement expressions, conditionals and remote references. Expressions are transmitted over the wire using a bytecode representation. Each operation is encoded as a byte with two fields. The upper 3 bits indicate the type of expression (currently Bool, Word8, Word16, or Word32) and the lower 5 bits indicate the operation type (literal, remote reference, addition, etc.). Expression operations may take one, two, or three parameters determined by the operation type, and each of the parameters is again an expression. Evaluation of the expression occurs recursively, until a terminating expression type of a literal, remote reference, or remote bind is reached. Figure 3 shows an example of encoding the addition of Word8 literal value of 4 the first remote reference defined on the board, as well as a diagram of that expression being used in an analogWrite command.

Conditionals are packaged in a similar manner to the way tasks are packaged, with the commands and procedures packaged into a code block. Two code blocks are used for the IfThenElse conditional (one block for the then branch, and one for the else branch), and one code block is used for the While loop. In

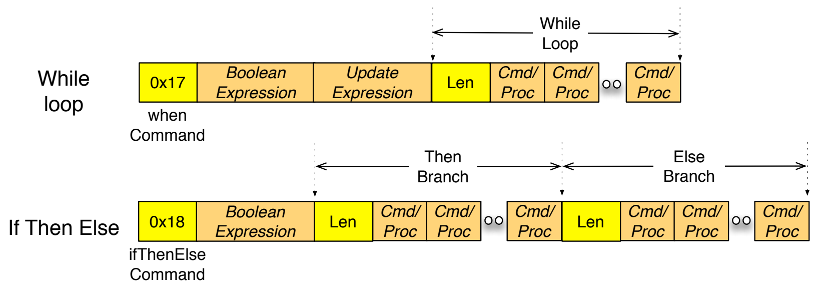


Fig. 4. Protocol Packing of Conditionals

addition, a byte is used for each code block to indicate the size of the block. A current limitation of conditionals in the protocol is that the entire conditional and code blocks must fit within a single Haskino protocol frame. However, if the conditional is part of a task, this limitation does not apply, as a task body may span multiple Haskino protocol frames. Figure 4 shows the encoding of both conditional types.

Now that we have described the components of the deeply embedded version of Haskino, we can return to a deep version of the simple example we used earlier.

```
exampleE :: IO ()
exampleE = withArduino True "/dev/cu.usbmodem1421" $ do
           let button = 2
           let led1 = 6
           let led2 = 7
           x <- newRemoteRef false
           setPinModeE button INPUT
           setPinModeE led1 OUTPUT
           setPinModeE led2 OUTPUT
           loopE $ do
                writeRemoteRef x  =<< digitalReadE button
                ex <- readRemoteRef x
                digitalWriteE led1 ex
                digitalWriteE led2 (notB ex)
                delayMillis 100
```

This deep example looks very similar to the strong example in structure. The binding x, which was previously stored on the host, is now kept on the Arduino, and created by the `newRemoteRef` function. The `writeRemoteRef` function updates the remote reference and is passed an expression, which in this case is the result of a remote computation using the `=<<` operator. The remote binds represented in this `writeRemoteRef` example, and the bind to `ex` with the value returned from `readRemoteRef` in the following line require implicit allocation on the Arduino. The Haskino firmware currently implements this allocation by storing the result of a procedure computation that would normally be sent across the serial interface to a local buffer associated with that bind instance instead. The expression bytecode language includes an `EXPR_BIND` operator, which takes it's input from this local buffer. Determining the best method of dealing with these allocations is still an open issue in our research.

In this example, since the computation results that are stored in the remote reference are used only within one iteration of the loop, the RemoteRef is not strictly required, but is used to demonstrate the RemoteRef API. The loop body could have been written using only binds as shown below.

```
loopE $ do
     ex <- digitalReadE button
     digitalWriteE led1 ex
     digitalWriteE led2 (notB ex)
     delayMillis 100
```

The tasks discussed in Section 4 become much more useful with the deeply embedded implementation. In the strong implementation, they were limited to sequences of commands and delays, as the remote language had no method of either binding computations together, or storing the result of a computation for future use. However, with the deep implementation, tasks may use the procedure primitives as well, and with the addition of conditionals, full programs may be stored for execution at a later time.

7 Comparing Runtime-Tethered Strong to Deeply Embedded Strong Remote Monad

Table 1 summarizes the major differences we found between the strong and deep implementations. In the strong version, all values are stored on the host, and passing values between computations requires communication with the host. With the deep version, values may be stored on the Arduino and passed between computations on the Arduino, eliminating the need for intermediate host communications. The basic task scheduling mechanism is able to use the full power of the language in the deep version, where it is limited to only commands with the strong version. One limiting factor of the deep version, is that the size of the program that may be written is limited by the available Arduino memory, while the strong version, due to the host interaction, is only limited by the larger host memory.

8 Cutting the Cord

One final addition to the firmware and Haskino language has allowed us to reach our goal of executing a stored Haskino program on the Arduino without requiring a connection to the host. The addition of the bootTaskE primitive allows the programer to write one previously defined task to EEPROM storage

Table 1. Comparison of Strong and Deep Embedding

	Runtime-tethered	**Deeply-embedded**
Values Stored On	Host	Arduino
Binds Occur On	Host	Arduino
Conditionals on Target	No	Yes
Tasks Can Use Procedures	No	Yes
Maximum Program Size	Limited by Host Memory	Limited by Arduino Memory
Communication Overhead	Higher	Lower

on the Haskino. The Haskino firmware checks for the presence of a boot task during the boot process, and if it is present, copies the task from EEPROM to RAM, and starts it's execution.

The following example illustrates how a programmer would create a boot task on the Arduino. The functionality of the program is the same as our other button and 2 LED examples. In this case, the `createTaskE` primitive is used to create the task in RAM on the Arduino, using the program stored in the example monad. The `bootTaskE` function is then called to write the task from RAM to EEPROM. On the next power on, the interpreter will start execution of the task. The `scheduleReset` primitive may be used to clear a previously written program from EEPROM.

```
example :: Arduino ()
example = do let button = 2
             let led1 = 6
             let led2 = 7
             x <- newRemoteRef (lit False)
             setPinModeE button INPUT
             setPinModeE led1 OUTPUT
             setPinModeE led2 OUTPUT
             loopE $ do
                  writeRemoteRef x  =<< digitalReadE button
                  ex <- readRemoteRef x
                  digitalWriteE led1 ex
                  digitalWriteE led2 (notB ex)
                  delayMillis 100

exampleProg :: IO ()
exampleProg = withArduino False "/dev/cu.usbmodem1421" $ do
              let tid = 1
              createTaskE tid example
              bootTaskE tid
```

We have achieved our original goal of programming a stand alone Arduino, using Haskell. The remote monad design pattern served us well by providing a path to this stand alone solution. There is however, much more that can be done. Having the ability to generate code from a DSL opens many possibilities. For example, recompiling to bake in timing, security contraints, or robustness concerns are possible paths forward.

9 Related Work

There is other ongoing work on using functional languages to program embedded systems in general, and the Arduino in specific. An early use of deep embeddings for remote execution was in the domain of graphics [5, 6]. A recent example is the Ivory language [7] provides a deeply embedded DSL for use in programming high assurance systems, but does not make use of the strong remote monad design pattern, and generates C rather than use a remote interpreter.

The Feldspar project [8–10] is Haskell embedding of a monadic interface that targets C, and focuses on high-performance. Interestingly, this work also attempt to make use of both deep and shallow embeddings inside a single implementation. Both Feldspar and Haskino use some form of monadic reification technology [11–13].

There is ongoing, and as yet unpublished, work by Pieter Koopman and Rinus Plasmeijer at Radboud University Institute for Computing and Information Sciences to develop a deep embedding around a Clean-based deep DSL. Their work also does not use the remote monad, and generates C rather that use a remote interpreter. Also, Kiwamu Okabe of Metasepi Design and Hongwei Xi of Boston University, in an as yet unpublished work, use a direct language implementation of ATS, as opposed to a DSL, to program the Arduino.

10 Conclusion and Future Work

Our two ways of structuring remote computations, provide complimentary but effective ways of using Haskell as a development environment for Arduino software. The strong Haskino provides a method for quick prototyping of software in a tethered environment. The deep version of Haskino allows the programmer to bring the full power of Haskell to developing standalone software for the Arduino. The connected version need not be limited to serial connections, as the Arduino Stream class would allow Arduino Ethernet connections to be used in a similar manner.

In the future, we plan to add a third way to our methodology, and directly generate C programs from our Arduino Monad. This will allow us to bootstrap the system. We also want to extend the scheduling mechanisms in Haskino, using the task structure for interrupt processing, and adding mechanism for communications between tasks in the system. Finally, we also plan on investigating using HERMIT [14] to semi-automatically translate from programs written in the tethered strong remote monad into programs written using the deep embedding. This will improve the applicability of the library.

This material is based upon work supported by the National Science Foundation under Grant No. 1350901.

References

1. Erkok, L.: Hackage package `hArduino-0.9` (2014)
2. Gill, A., Sculthorpe, N., Dawson, J., Eskilson, A., Farmer, A., Grebe, M., Rosenbluth, J., Scott, R., Stanton, J.: The remote monad design pattern. In: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, ACM (2015) 59–70
3. Steiner, H.C.: Firmata: Towards making microcontrollers act like extensions of the computer. In: New Interfaces for Musical Expression. (2009) 125–130
4. Elliott, C.: Hackage package `boolean-0.2.3` (2013)
5. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming. (1997)
6. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(2) (2003)
7. Elliott, T., Pike, L., Winwood, S., Hickey, P., Bielman, J., Sharp, J., Seidel, E., Launchbury, J.: Guilt free ivory. In: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, ACM (2015) 189–200
8. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajdax, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: MEMOCODE'10. (2010) 169–178
9. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of feldspar. In: Implementation and Application of Functional Languages. Springer (2011) 121–136
10. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: Trends in Functional Programming. Springer (2013) 21–36
11. Persson, A., Axelsson, E., Svenningsson, J.: Generic monadic constructs for embedded languages. In Gill, A., Hage, J., eds.: Implementation and Application of Functional Languages. Volume 7257 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 85–99
12. Svenningsson, J.D., Svensson, B.J.: Simple and compositional reification of monadic embedded languages. In: Proceedings of the 18th International Conference on Functional Programming, ACM (2013) 299–304
13. Sculthorpe, N., Bracker, J., Giorgidze, G., Gill, A.: The constrained-monad problem. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ACM (2013) 287–298
14. Farmer, A., Sculthorpe, N., Gill, A.: Reasoning with the HERMIT: tool support for equational reasoning on GHC core programs. In: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, ACM (2015) 23–34