# Threading the Arduino with Haskell

Mark Grebe and Andy Gill

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

**Abstract.** Programming of embedded microcontrollers often requires the scheduling of independent threads of execution, specifying the interaction and sequencing of actions in the multiple threads. Developing and debugging such multi-threaded systems can be especially challenging in highly resource constrained systems such as the Arduino line of microcontroller boards. The Haskino library, developed at the University of Kansas, allows programmers to develop code for Arduino-based microcontrollers using monadic Haskell program fragments. This paper describes our efforts to extend the Haskino library to translate monadic Haskell code to multi-threaded code executing on Arduino boards.

**Keywords:** Haskell, Arduino, Remote Monad, Embedded Systems, Scheduling

## 1 Introduction

The Haskino library was written to advance the use of Haskell to program systems based on the Arduino line of micocontrollers. Software written for embedded microcontrollers routinely requires multiple threads of execution to efficiently and easily meet its requirements. The original version of Haskino [1] supported only single threaded, run to completion style programming. We have since extended Haskino to support multi-threaded, concurrent operation.

In this paper, we discuss the following.

- We describe how threads are implemented in Haskino, and how it handles scheduling operations.
- We discuss the methods of inter-thread communication used in Haskino.
- We describe the Haskino code generator, which generates C code from the Haskino DSL, allowing native compilation on the Arduino in addition to interpretation.
- We discuss other additions that have been made to the Haskino DSL since the original version.
- We explain the additions that have been made to Haskino to support quick prototyping and easy debugging of multi-threaded programs.
- We then instantiate these ideas into a number of concrete examples.

### 1.1 Background

The Haskino library has its roots in the hArduino package [2], written by Levent Erkök, which allows programmers to control Arduino boards through a serial connection. In hArduino, the serial protocol used between the host computer and the Arduino, and the firmware which runs on the Arduino, are together known as Firmata [3].

The original version of Haskino, extended hArduino by applying the concepts of the strong remote monad design pattern [4] to provide a more efficient way of communicating, and generalizing the controls over the remote execution. In addition, it added a deep embedding, control structures, an expression language, and a redesigned firmware interpreter to enable standalone software for the Arduino to be developed using the full power of Haskell.

The remote monad design pattern splits primitives to be executed on the remote Arduino into commands and procedures. Commands are primitives that do not return a result, and do not have a temporal consequence, such as executing a delay. Procedures are remote primitives that do return a result, or do have a temporal consequence. The remote monad design pattern then uses bundling strategies to combine the commands and procedures efficiently, as they are sent to the remote Arduino.

## 2 Haskino Threads

The original 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. However, as they have no way to store results from one procedure for use in another command or procedure, the usefulness of these tasks is severely limited.

The initial version of Haskino extended the ability of tasks, allocating remote binds to store the results of a procedure and use the result in a subsequent command or procedure. It was, however, still limited to running a single task to completion.

We have subsequently extended Haskino to allow it to handle multiple threads of execution, with communication between the threads, and cooperative multitasking.

As an example of multiple threads running in a Haskino program, we present the following program.

```
blinkDelay :: Expr Word32
blinkDelay = 125

taskDelay :: Expr Word32
taskDelay = 2000

semId :: Expr Word8
semId = 0
```

```
myTask1 :: Expr Word8 -> Arduino ()
myTask1 led = 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

myTask2 :: Arduino ()
myTask2 = do
    loopE $ do
        giveSemE semId
        delayMillisE taskDelay

initExample :: Arduino ()
initExample = do
    let led = 13
    -- Create the tasks
    createTaskE 1 $ myTask1 led
    createTaskE 2 myTask2
    -- Schedule the tasks to start in 1 second,
    -- the second starting after the first
    scheduleTaskE 1 1000
    scheduleTaskE 2 1050

semExample :: IO ()
semExample = withArduino True "/dev/cu.usbmodem1421" $ do
    initExample
```

This example creates two tasks. The first task, myTask1, sets the mode of the LED pin to output, then goes into an infinite loop. Inside the loop, it takes a semaphore, and when the semaphore is available it blinks the LED rapidly three times. The second task, myTask2, executes an infinite loop where it gives the semaphore, then delays for two seconds. The main function, semExample, creates the two tasks and schedules them to execute, using the Arduino connected to the specified serial port.

## 3 Scheduling the Interpreter

To enable scheduling in Haskino, the Haskino firmware interpreter required modification to allow another task to run when the currently executing task was suspended due to a delay or a wait on a resource. The scheduler in the Firmata firmware only ran tasks to completion, so no interruption and resumption of tasks was allowed. The scheduler in the initial version of the Haskino interpreter was modeled after that scheduler, and therefore was limited to run to completion tasks as well.

The deeply embedded version of Haskino defined three types of conditional monadic structures, an If-Then-Else structure, a While structure, and an infinite LoopE structure. The current version of Haskino has added a ForIn structure, which allows a monadic computation to be performed over each element of a list of 8 bit words. Each of these structures contains the concept of execution of basic code blocks within the interpreter. For example, execution of the LoopE structure consists of executing the same basic block of code over and over, while the If-Then-Else structure consists of execution of either the basic block that makes up the Then leg of the structure, or the basic block associated with the Else leg of the structure.

To allow the scheduler to interrupt execution of a basic block in a task, and then later restore execution when the task resumes, a method for saving and restoring the execution state of the task is required. In an operating system, this is normally done by saving and restoring the processor's registers, as well as giving each thread its own stack. In the Haskino interpreter, each task has it's own context, which provides the storage for the bound variables. This corresponds to the separate stack for each thread in a traditional operating system.

In addition, the interpreter must also store information in the context which indicates where in the basic block execution of the task was interrupted, such that it may be restored when the task resumes. For all of the control structures containing basic blocks, the location (in bytes from the start of the block) of the command or procedure that was executing when the interruption occurred is stored. For the simplest of the control structures, ForE and While, this is all that is required. The other two control structures require an additional piece of information to be stored. The If-Then-Else structure requires storing which branch code block was being executed, either the Then branch or the If branch. The ForIn structure requires storing an index indicating which element of the list the code block was being executed for.

As the Haskino control structures may be nested, the scheduler is required to keep track of not just the state of execution in one basic block, but instead must track the state of execution in a stack of basic blocks leading up to the point that code execution was suspended. When the task is later resumed, the interpreter must walk that stack in reverse order, restoring the state of the task for each of the other nested basic blocks.

Currently, there are two procedures which cause the Haskino scheduler to interrupt the execution of a task, and potentially start execution of another. The first of these procedures is the delayMillis command, which will delay a task for specified number of milliseconds. When the procedure is executed, the state of current task is saved, and the time when the task should resume execution is stored in the task's context. The scheduler then checks if another task is ready to run, based on its next execution time having passed, or a resource it was waiting on having become available. If a ready to run task is found, it's state of execution is restored by the method previously discussed, and it's execution is resumed. A second delay procedure, delayMicros, exists for those cases where the programmer wishes a task to have a short delay without the possibility of being interrupted. The second procedure which can cause a reschedule is taking a semaphore, which is described in the following section.

## 4 Inter-thread communication

Running multiple threads of computation is of limited use if the threads do not have a method of communicating with each other. To enable communication and syncronization between tasks, Haskino provides several methods. First, the RemoteReference class provides atomic storage methods that can be used to pass data between Haskino tasks.

Haskino also provides binary semaphores for synchronization between Haskino tasks. A binary semaphore may be given by one task by issuing the giveSem procedure, while the task that wants to synchronize with the first task can do so by issuing the takeSem procedure. When a task issues a takeSem procedure, and the binary semaphore that it refers to is not available, the task will be suspended. When another task later makes the semaphore available through a giveSem procedure, the scheduler will then make the task taking the semaphore ready to run. If the binary semaphore is available when takeSem is called, the semaphore is made unavailable, but the task is not suspended, but continues operation. In addition, if a task is already waiting on an unavailable semaphore when another task calls giveSem, the semaphore is left unavailable, but the task waiting on it is made ready to run.

The inclusion of binary semaphores also enables Haskino to handle another important aspect of programming embedded systems, the processing of interrupts. In addition to handling multiple tasks, the Arduino monadic structures may also be attached to handle external Arudino interrupts. For example, the following example uses a simple interrupt handler which gives a semaphore to communicate with a task. It is similar to our earlier two task example, but in this case, the interrupt handling task is attached to the interrupt using the attachIntE command, which specifies the pin of the interrupt to attach to.

```
blinkDelay :: Expr Word32
blinkDelay = 125

semId :: Expr Word8
semId = 0
```

```
myTask :: Expr Word8 -> Arduino ()
myTask led =
    loopE $ do
        takeSemE semId
        digitalWriteE led true
        delayMillisE blinkDelay
        digitalWriteE led false
        delayMillisE blinkDelay

intTask :: Arduino ()
intTask = giveSemE semId

initIntExample :: Arduino ()
initIntExample = do
    let led = 13
    setPinModeE led OUTPUT
    let button = 2
    setPinModeE button INPUT
    let myTaskId = 1
    let intTaskId = 2
    createTaskE myTaskId (myTask led)
    createTaskE intTaskId intTask
    scheduleTaskE myTaskId 50
    attachIntE button intTaskId FALLING

intExample :: IO ()
intExample = withArduino True "/dev/cu.usbmodem1421" $ do
    initIntExample
```

## 5 Code Generation

The interpreted version of the Haskino DSL provides a quick turnaround Arduino development environment, including features for easy debugging. However, it has one major disadvantage. The interpreter takes up a large percentage of the flash program storage space on the smaller capability Arduino boards such as the Uno. The only other resource on such boards for storing interpreted programs to be executed when the Arduino is not tethered to a host computer is EEPROM, which is what the current interpreter uses. However, this resource is also relatively small (1K byte) on these boards. These storage limitations directly limit the complexity of programs which can be developed using the interpreted version of Haskino when not connected to a host computer.

To overcome these limitations, we have developed a compiler that translates the same Haskell DSL source code used to drive the interpreter, into C code. The C code may then be compiled and linked with a C based runtime which is much smaller than the interpreter. The compiler takes as input the same Arduino monad that is used as input to the withArduino function to run the interpreter, and the file to write the C code to.

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

Executing the compileProgram function on the initExample monad from the semaphore example in Section 2, produces the following C code. We will explain specific portions of the generated code in the following sections.

```
#include "HaskinoRuntime.h"

void haskinoMain();
#define HASKINOMAIN_STACK_SIZE 100
byte haskinoMainTcb[sizeof(TCB) + HASKINOMAIN_STACK_SIZE];
void task2();
#define TASK2_STACK_SIZE 104
byte task2Tcb[sizeof(TCB) + TASK2_STACK_SIZE];
void task1();
#define TASK1_STACK_SIZE 100
byte task1Tcb[sizeof(TCB) + TASK1_STACK_SIZE];

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

void loop()
    {
    }

uint8_t ref0;
uint8_t ref1;

void task1()
    {
    pinMode(13,1);
    ref1 = 0;
    while (1)
        {
        takeSem(0);
        ref1 = 0;
        for (;(ref1 < 3);ref1 = (ref1 + 1))
            {
```

```
            digitalWrite(13,1);
            delayMilliseconds(125);
            digitalWrite(13,0);
            delayMilliseconds(125);
            }
        }
    taskComplete();
    }

void task2()
    {
    ref0 = 0;
    while (1)
        {
        uint8_t bind0;

        giveSem(0);
        bind0 = ref0;
        ref0 = (bind0 + 1);
        debug(showWord8(bind0));
        delayMilliseconds(2000);
        }
    taskComplete();
    }

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

### 5.1 Initialization

Arduino programs consist of two main functions, setup(), which performs the required application initialization, and loop(), which is called continuously in a loop for the main application. For Haskino applications, any looping is handled inside of the monadic Haskino code, and the compiled code uses only the setup() function. The loop() function is left empty, and is only provided to satisfy the link requirement of the Arduino library.

The setup() function serves three purposes. First, it initializes the memory management of the Haskino runtime, which is described in Section 5.3. Second, it creates the initial root task of the application. The compiler generates the code associated with the main monadic function passed to compileMonad as the C function haskinoMain(). The steup() function creates the initial task by calling createTask(), passing a pointer haskinoMain(), and schedules the task to start immediately by calling scheduleTask(). Finally, the runtime scheduler, described in Section 5.4, is started by calling the startScheduler() function.

### 5.2 Storage Allocations

Three types of storage are allocated by the compiler. RemoteReference's are compiled into global C variables, named refX, where X is the id of the remote reference. In the example, two Word8 remote references are used, and their newRemoteRef calls are compiled into the following allocations:

```
uint8_t ref0;
uint8_t ref1;
```

Binds in the Haskino DSL are compiled into local variables, and are therefore allocated on the stack. The number of binds for each code block is tracked by the compiler, and the binds are defined local to the code block in which they are used. In the example, there is one Word8 bind in task2, used inside of the while loop:

```
    while (1)
        {
        uint8_t bind0;

        giveSem(0);
        bind0 = ref0;
        ref0 = (bind0 + 1);
        debug(showWord8(bind0));
```

Like the tasks in the interpreter, each task in the compiled code requires a context to track its state. In the compiled code, this context consists of the C stack, as well as several other state variables, such as the next time the task should run and a flag indicating if the task is blocked. Together, these make up the task control block (TCB) for the task. The compiler allocates space for the task control block statically, sizing the block based on the size of the fixed elements of the block, a default amount of stack space to account for Arduino library usage, and finally stack space for the number of binds used by the task, which the compiler tracks while generating the code. The following shows the generated code used to define the TCB for task1 from the semaphore example, as well as how the TCB and stack size are passed to the task creation function.

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

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

### 5.3 Dynamic Memory Management

Both the Haskino interpreter, and the compiler require some form of dynamic memory management to handle the Word8 list expressions which are used in the Haskino expression language for strings and byte array data such as I2C input and output (discussed in Section 6). In both cases the garbage collection scheme is simple, with memory elements being freed when an associated reference count for the element goes to zero. The interpreter uses the standard libc memory routines malloc() and free(), which allocates space from the heap.

The libc heap allocation scheme was not practical for use with the generated thread code. With the standard Arduino libc memory management, the programs stack grows down from the top of memory, while the heap grows up from the bottom of available memory. The malloc() routine includes a test to make sure that the new memory allocation will not cause the heap to grow above the stack pointer. While this will work with the interpreter, the compiler statically allocates the stack for each of the tasks, and the stack pointer for all of the tasks would then be below the heap, causing any memory allocation to fail.

One possible solution to this issue that was considered was to rewrite the Arduino memory management library to remove the heap/stack collision detection, so that it would be usable with multiple stacks. Instead, to improve speed and determinism of the memory allocation and garage collection in the compiled code, a fixed block allocation scheme was instead chosen. Through a library header file, the programmer is able to choose the number of 16, 32, 64, 128 and 256 byte blocks available for allocation. The runtime then keeps a linked list of the free blocks for each block size, and the memory allocator simple returns the head of the free list of the smallest block size larger the the requested size. If no blocks of that size are available, then the next larger free list is tried until a free block is found, or until the allocation fails.

### 5.4 Scheduling the Generated Code

The small Haskino runtime system used with the generated C code needs to duplicate the scheduling capabilities of the Haskino interpreter, to allow Haskino programs to be move seamlessly between the two environments. These capabilities are provided by a small multitasking kernel that is a core part of the runtime. Like the Haskino interpreter, generated tasks are cooperative, only yielding the processor at delays and semaphore takes.

The memory allocation of the task control blocks was discussed in the Section 5.3. The scheduling algorithm used is a simple cooperative algorithm. Since the number of tasks expected is relatively small, a separate ready list is not used. Instead, each time the scheduler is run when a task yields the processor, the list of all tasks is scanned starting at the task after the yielding task for a task whose next time to run is less than or equal to the current time, and is not blocked. Starting the list search at the next task after the yielding task ensures that scheduling will occur in a round robin sequence of the ready tasks, even if each tasks yields with a delayMilliseconds(0).

The compiler inserts a taskComplete() call at the end of each generated task. If the task ever reaches this call, it will mark the task as blocked so that it will no longer run. As task control blocks are allocated statically, the task control block memory is not freed.

## 6 Other Additions to Haskino

The original deep embedding of the Haskino DSL and interpreter provided the capability to handle typed expressions of boolean and unsigned integers of length 8, 16 and 32 bits. This covered the types used by the basic digital and analog input and output functions in the Arduino API.

However, to extend the DSL for more complex libraries such as the stepper motor, servo motor, and I2C libraries, the handling of signed types, floating points, and lists of unsigned 8 bit words has been added to the Haskino DSL expression language. In addition to adding float data types and their basic operations, the expression language was also expanded to include most of the standard math library primitives, including trigonometric functions. Primitives to convert between numeric data types, including toInteger, fromInteger, trunc, round, frac, ceil, and floor have also been added.

Handling reads and writes from I2C devices, as well as displaying text on LCD and other displays, requires the ability to handle a type for a collection of bytes. As Haskino is a Haskell DSL, the intuitive choice for the collection is a list of Word8. In the new version, Haskino's expression language has been enhanced with primitives for cons, append, length, and element operations on expressions of [Word8]. In addition, show primitives have been added to convert other types into lists of Word8.

## 7 Debugging

The original version of the Haskino DSL provided rudimentary debugging capabilities through a debug local which made use of host Haskell show functions:

```
debug :: String -> Arduino ()
```

However, since the debug parameters were evaluated locally on the host, not in the Haskino interpreter, it could not be used for debugging intermediate results within deeply embedded conditionals or loops, or for debugging within tasks. The current version of the Haskino DSL instead includes a debug Procedure whose expression parameters are evaluated on the Arduino:

```
debugE :: Expr [Word8] -> Arduino ()
```

The evaluated expression is returned to the host via the Haskino protocol, and the message displayed on the host console. It can make use of the show primitives added to the expression language to display the values of remote references or remote binds.

An additional procedure was also added to the DSL language, debugListen, which keeps the communication channel open listening for debug messages. This was required as the channel is normally closed after the last command or procedure has been sent. If the last command is a loop or task scheduling, this procedure may be used to ensure that debug messages are received and displayed on the host while the loop or task executes on the host.

One of the key features of Haskino is that the same monadic code may be used for both interpreted and compiled versions. This allows for quick prototyping with the tethered, interpreted version, and then compiling the code for deployment. This duality of environments is supported with the debugging primitives as well. When compiled, the debugE procedure will output the evaluated byte list to the serial port, allowing the same debug output displayed by the interpreted version to be used in debugging the compiled version as well.

## 8 Examples

To better illustrate the utility of the Haskino system with a multithreading program, we present the following slightly more complex example. In this example, an Arduino board has multiple LED lights connected to it (in the example code below, three lights), and each of these lights are required to blink at a different, constant rate.

The basic monadic function for blinking a LED is defined as ledTask, which is parameterized over the pin number the LED is connected to, and the amount of time in milliseconds the LED should be on and off for each cycle. This function sets the specified pin to output mode, then enters an infinite loops turning the LED on, delaying the specified time, turning the LED off, and then again delaying the specified time.

The main function of the program, initExample, creates three Hakino tasks, each with a different LED pin number, and a different delay rate. The three created tasks are then scheduled to start at a time in the future that is twice their delay time. The task with an ID of 1 will be the first to run, as it scheduled to start at the nearest time in the future (1000 ms). It will run until it reaches its first call to delayMillisE. At that point, the scheduler will be called. The scheduler will reschedule task 1 to start again in 500ms, and as no other tasks will yet be started at that time then call the Arduino delay() function with the same time delay. When the delay() function returns, task 1 will be the only task ready to run, so it will run again until it reaches the second second delayMillisE call, when the scheduler will be called and will delay() as before. When delay() returns the second time both task 1 and task 2 will be ready to run. Since task 1 was the last to run, the scheduler will search the task list starting at the task after task 1, and will find task 2 ready to run, and it will be started. Task 2 will run until it reaches the delay, at which point the scheduler will be called, and it will restart task 1 since it was also ready to run. This process will continue, with each task running (turning it's LED on or off) until it reaches a delay, at which point it will cooperatively give up its control of the processor and allow another task to run.

The final two functions in the example, ledExample and compile are used to run the initExample monad with the interpreter and compiler respectively.

This example demonstrates the ability to write a program where using multiple threads to implement concurrency greatly simplifies the task. This code could have been written with straight inline code, but would require calculating the interleaving of the delays for the various LED's. However, in that straight line code it would be more difficult to expand the number of LEDs, or to handle staggered start times. Both of those cases are easily handled by the multithreaded code, and the amount of code is also smaller in the multithreaded case, since the ledTask function is reused.

```
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
    -- Create the tasks
    createTaskE 1 $ ledTask led1  500
    createTaskE 2 $ ledTask led2 1000
    createTaskE 3 $ ledTask led3 2000
    -- Schedule the tasks to start in 1, 2, and 4 seconds respectively
    scheduleTaskE 1 1000
    scheduleTaskE 2 2000
    scheduleTaskE 3 4000

-- Execute this function to run program with firmware interpreter
ledExample :: IO ()
ledExample = withArduino True "/dev/cu.usbmodem1421" $ do
    initExample

-- Execute this function to generate C code to be used with the runtime.
compile :: IO ()
compile = compileProgram initExample "multiLED.ino"
```

## 9 Related Work

There is other ongoing work on using functional languages to program embedded systems in general, and the Arduino in specific. A shallowly embedded DSL for programming the Arduino in the Clean language, called ArDSL has been developed [5]. Their work does not make use of the remote monad design pattern, and does not provide a tethered, interpreted mode of operation.

The Ivory language [6][7] provides a deeply embedded DSL for use in programming high assurance systems. It also does not make use of the strong remote monad design pattern, and generates C rather than use a remote interpreter. An additional EDSL built on top of Ivory, called Tower [6], provides the ability to define tasking for multithreaded systems. However, it depends on the support of an underlying RTOS, as opposed to the minimal scheduler of Haskino.

The frp-arduino [8] provides a method of programming the Arduino using Haskell, but using a functional reactive programming paradigm, and once again only compiling to C code.

## 10 Conclusion and Future Work

The updated Haskino provides two complimentary ways of developing multithreaded software for the Arduino line of embedded development boards. First, the Haskino interpreter may be used for fast prototyping while attached to a host computer. Then, the same monadic Haskell code may be compiled using the Haskino compiler, allowing for standalone execution with a smaller memory footprint.

In the future, we also plan on investigating using HERMIT [9] 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. This will improve the applicability of the library. We would also like to expand the scheduler in Haskino, adding priorities and preemption to the current cooperative multithreading.

## 11 Acknowledgment

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

## References

1. Grebe, M., Gill, A.: Haskino: A remote monad for programming the arduino. In: Practical Aspects of Declarative Languages, Springer (2016) 153–168
2. Erkok, L.: Hackage package `hArduino-0.9` (2014)
3. Steiner, H.C.: Firmata: Towards making microcontrollers act like extensions of the computer. In: New Interfaces for Musical Expression. (2009) 125–130

4. 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
5. Koopman, P., Plasmeijer, R.: A shallow embedded type safe extendable dsl for the arduino. In: Trends in Functional Programming, Springer (2015) 104–123
6. Hickey, P.C., Pike, L., Elliott, T., Bielman, J., Launchbury, J.: Building embedded systems with embedded dsls. In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, ACM (2014) 3–9
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. Lindberg, R.: Hackage package `frp-arduino-0.1.0.3` (2015)
9. 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