Embedded Programming with Embedded Domain-Specific Languages (EDSLs) in Haskell

Eliane I. Schmidli
OST – Eastern Switzerland University of Applied Sciences
MSE Seminar “Programming Languages”
Supervisor: Prof. Dr. Farhad Mehta
Spring 2023

Abstract

Embedded domain-specific languages (EDSLs) make it possible to program embedded systems with Haskell. Programming in Haskell has many advantages such as the reduction of runtime errors and writing code more elegantly. The Haskell library Haskino provides two EDSL-based variants for programming Arduinos. In the first variant, commands are sent interactively to an Arduino that is connected to the computer. The second variant turns the Haskell code into an abstract syntax tree (AST) representing actions and computations. This structure can be used to generate C code that can be executed directly on the Arduino. The advantage of Haskell becomes apparent when using paradigms such as functional reactive programming (FRP). In the first variant offered by Haskino, FRP can be used without any problems because it is executed in the Haskell environment. In the second, it would be necessary to integrate the FRP functionality into the AST and to define the translation into C. The ability to run FRP code directly on Arduino is demonstrated by the frp-arduino library. This paper introduces EDSLs in Haskell, their application in the Haskino library, and shows an Arduino programming example using Haskell and FRP.

1 Introduction

Safety and correctness are essential in embedded systems. Due to Haskell’s strong, static type system, the functional programming language can help to avoid errors. Furthermore, the use of Haskell can lead to more modular, elegant code, which can improve maintainability. Unfortunately, it is not possible to run Haskell directly on small embedded systems with limited resources, such as an Arduino.

An alternative is to use an EDSL. This is a domain-specific language (DSL) embedded in a host programming language such as Haskell. In the subsequent sections, an overview of EDSLs is provided. Section 2 presents an EDSL that enables programming the Arduino in Haskell. Section 3 shows the benefits of using Haskell for embedded programming by controlling the Arduino with FRP.

1.1 Embedded and External DSL

A DSL is a computer language designed for a specific problem. Unlike general-purpose languages, it uses terms and syntax that are close to the problem domain. An example is the Very High Speed Integrated Circuit Hardware Description Language (VHDL) used for circuit design.

Two types of DSLs can be differentiated: internal DSLs (referred to as embedded DSLs (EDSLs)) and external DSLs. EDSL is a language embedded in a host language. So, it can use a subset of the features of the host language, such as its appearance and semantics. An external DSL, on the other hand, is a language created from scratch (like VHDL). To process it, a complete parser must be written [Fow19].

1.2 Shallow and Deep EDSL

There are two different types of EDSLs; shallowly embedded and deeply embedded. The shallow EDSL uses elements and the runtime environment of the host language. Therefore the syntax and semantics of a shallow EDSL are close to those of the host language. Deep EDSLs, on the other hand, do not use the host language for direct calculations, but to generate a new language. The resulting output is an AST representing actions or calculations to be executed. Its interpretation must be explicitly defined. Thus, it is possible to interpret and execute the construct in a language other than the host language [GYG17].

For example, in a shallow EDSL, an integer type can be defined as a type synonym:

```haskell
type AInt = Int
```

In shallow EDSL, existing Haskell functions accepting arguments of type `Int` can also be called with values of type `AInt` (A was chosen for Arduino). For example, functions of class `Num` like the addition with `(+)`. So, in shallow EDSLs, values like `myAddition` are computed directly.

```haskell
myAddition :: AInt
myAddition = ( 1 + 2 ) + 3
```

In contrast, when using a deep embedding, a new language which describes the actions or computations that need to be performed is defined. In the case of our current example, the following type is needed:

```
data Expr = Add Expr Expr | Val Int
```

Using the type Expr results in a tree structure describing the calculations to be performed. Here, the expression myAdditionE represents the structure of myAddition, but cannot be calculated directly anymore.

```
myAdditionE :: Expr
myAdditionE = Add (Add (Val 1) (Val 2)) (Val 3)
```

To interpret the function in Haskell, a function eval is needed that evaluates the two expression type variables of Add recursively and then adds the resulting integers together.

```
evalE :: Expr -> Int
evalE Val x = x
evalE Add x y = (eval x) + (eval y)
```

But it would also be possible to translate the resulting structure into another programming language [Aug12].

1.3 Embedded Programming with Haskell

There are several libraries and languages that support programming close to hardware with Haskell. One of these is CλasH; a functional hardware description language that generates code in VHDL [Baa09]. There are also deep EDSLs that generate C code, such as Co-Pilot [PWNG12] and Ivory [EPW15].

A library that takes advantage of both shallow and deep EDSL is Haskino. It allows programming the Arduino in Haskell[GYG17]. It is based on the shallow EDSL hArduino which requires a connection between the host and the Arduino [Erk22]. An alternative to deep EDSL is the method described in Compiling to Categories [Ell17].

The following sections introduce the use of shallow and deep EDSL in Haskino. Furthermore, the limitations of Haskino are shown when trying to program the Arduino in FRP style.

2 Haskino

Haskino takes advantage of both EDSL variants. It is possible to write the code in the easy-to-use shallow EDSL. The code can then either be executed in the Haskell environment interactively sending commands to the connected Arduino or compiled into C code by using a deep EDSL and executed directly on the Arduino. The first variant is suitable for debugging and prototyping. The second variant is more efficient and produces executables in smaller sizes [GYG17]. The two variants are described in more detail in the following sections.

2.1 Interpreted interactive system

Haskino's interpreted interactive system executes the shallow EDSL in the Haskell environment and sends commands to the Arduino connected via a USB cable. They provide a firmware interpreter Arduino program that must be loaded on the Arduino. This interpreter receives instructions from the computer via a serial link and executes the corresponding Arduino commands [GYG17].

The user can write a Haskell program using the shallow EDSL provided by the Haskino library. The function signatures are like those provided by Arduino. The similarity between the programs can be seen in the blink example, which is considered to be Arduino's Hello World program. It repeatedly turns the built-in LED on and off on boards such as UNO, MEGA, or ZERO.

In the Arduino programming language, two functions called setup and loop must be defined. The function setup is called in the beginning and can be used to initialize the input and output pins. The program that should be executed can be defined in the function loop. At runtime, a main loop calls this function at each iteration.

For the blink example [Ard23], setup initializes the digital pin LED_BUILTIN as an output.

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}
```

Then the program can be implemented in the loop function. The digitalWrite function sets the voltage level for the corresponding LED and accordingly turns it on (HIGH, resp. 1) or off (LOW, resp. 0). To wait one second between the switches, the function delay is executed with 1000 milliseconds.

```
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

The Haskell program with Haskino looks similar. It represents the Arduino using the RemoteMonad [GD16], an implementation of the concepts of the remote monad design pattern [GSD15]. It is a way of making calls to a remote target less expensive.

```
newtype Arduino a = Arduino (RemoteMonad
    ArduinoPrimitive a)
```

On boards such as UNO, MEGA, or ZERO, the pin number of the built-in LED is 13. The initialization is done before the loop. The constants HIGH and LOW are replaced by True and False.

```
blink :: Arduino ()
blink = do
   setPinMode 13 OUTPUT
   loop $ do
      digitalWrite 13 True
      delayMillis 1000
      digitalWrite 13 False
      delayMillis 1000
```

However, the execution of the shallow EDSL is different to the execution of the Arduino program. The loop runs on the computer in the Haskell environment. The instructions of this program are sent one by one to the interpreter on the Arduino. The interpreter executes the instruction and for the delay instruction sends back a message when the time is up. The following snippet of console output from the blink program shows the process.

```
[199:40444148161] Haskino:
    Sending:
    DIG_CMD_WRITE_PIN
        (EXPR_WORD8-EXPR_LIT 13)
        (EXPR_BOOL-EXPR_LIT True)

[200:40444148323] Haskino:
    Sending:
    BC_CMD_DELAY_MILLIS
        (Bind 0) <- (EXPR_WORD32-EXPR_LIT 1000)

[201:40444148475] Haskino:
    Waiting for response

[202:40444451142] Haskino:
    Received DelayResp

[203:40444451360] Haskino:
    Sending:
    DIG_CMD_WRITE_PIN
        (EXPR_WORD8-EXPR_LIT 13)
        (EXPR_BOOL-EXPR_LIT False)
```

2.2 Compiled System

To run a Haskino program directly on the Arduino, the program written in shallow EDSL can be automatically compiled into C by Haskino. The shallow EDSL is first translated by the Glasgow Haskell Compiler (GHC) into Haskino's deep EDSL and then compiled into a C program. The resulting Arduino file can be loaded onto the Arduino together with a runtime provided by Haskino [GYG17]. The following sections provide the intermediate translation steps for the blink example.

The shallow EDSL is first translated into the deep EDSL. To gain insight into the structure of Haskino's deep EDSL, the following section describes the loop body representation in the context of the blink example. A comprehensive explanation of the representation of control structures such as conditionals and loops in Haskino's deep EDSL can be found in [GYG17].

The operations digitalWrite and delayMillis are represented by the data structure ArduinoPrimitive. The Haskell types used in these operations are captured in the data structure Expr.

```
data Expr a where
    LitB   :: Bool -> Expr Bool
    LitW8  :: Word8 -> Expr Word8
    LitW32 :: Word32 -> Expr Word32

data ArduinoPrimitive :: * -> * where
    DigitalWriteE :: Expr Word8 -> Expr Bool ->
     ArduinoPrimitive (Expr ())
    DelayMillisE  :: Expr Word32 ->
     ArduinoPrimitive (Expr ())
```

The loop body representation in Haskino's deep EDSL is similar to that of the shallow EDSL, as shown below:

```
DigitalWriteE (LitW8 13) (LitB True)
DelayMillisE (LitW32 1000)
DigitalWriteE (LitW8 13) (LitB False)
DelayMillisE (LitW32 1000)
```

The deep EDSL representation of the blink example is then translated into the C programming language. The resulting C program looks like this:

```
#include "HaskinoRuntime.h"
// simplified
void setup() {
  haskinoMemInit();
  // 255 is the id of the task haskinoMain
  createTask(255, haskinoMain);
  scheduleTask(255, 0);
  startScheduler();
}

void haskinoMain() {
   pinMode(13,OUTPUT);
   while (1) {
      digitalWrite(13,1);
      delayMilliseconds(1000);
      digitalWrite(13,0);
      delayMilliseconds(1000);
  }
  taskComplete();
}
```

Unlike the original Arduino program, loop is not called here. Instead, a thread that executes the main program is started.

The setup function initializes the memory management of the Haskino runtime and creates and schedules a task executing the haskinoMain function. In haskinoMain, the pins that are used in the program are initialized. Then the infinite loop starts where the digitalWrite instruction of the Arduino programming language is called. The function delayMilliseconds reschedules the task by 1000 milliseconds. The scheduling functions are provided by the Haskino runtime and allow multi-threading [GG19]. When writing the program in the shallow EDSL, the functions createTask and scheduleTask can be used to run different tasks simultaneously. The next section provides an example using threads. It is only possible to use multi-threading with the compiled system in Haskino. The interactive mode does not provide this functionality.

2.3 Haskino Example with Threads

The use of threads in Haskino can be better seen in the following example. As in the blink example, the built-in LED should blink. Additionally, a microphone is connected to the Arduino, which has a digital output. On the microphone, a threshold for the volume can be set. A second LED is connected, which should light up when the threshold is exceeded.

For the first task, the same code as in the blink example can be used. This is written in a function called `ledTask`.

```haskell
led :: Word8
led = 13

ledTask :: Arduino ()
ledTask = do
    loop $ do
        digitalWrite led True
        delayMillis 1000
        digitalWrite led False
        delayMillis 1000
```

For the second task, `digitalRead` is used to read the value from the digital output of the microphone. In this example, the digital output is connected to pin three. If the volume threshold is exceeded, the value is `True`, otherwise it is `False`. A red LED is connected to pin nine and is turned on when the threshold is exceeded, otherwise it is turned off. On the last line, a small pause is inserted, so that the serial interface is not overloaded.

```haskell
ledRed :: Word8
ledRed = 9

digitalSoundPin :: Word8
digitalSoundPin = 3

soundTask :: Arduino ()
soundTask = do
    loop $ do
        digitalVal <- digitalRead digitalSoundPin
        if digitalVal
        then do
            digitalWrite ledRed True
        else do
            digitalWrite ledRed False
        delayMillis 50
```

In order to execute the program, the pin modes must first be defined. Here the two LEDs are outputs and the `digitalSoundPin` is an input. Now the two tasks can be created with `createTask`. Besides the function to be executed, an id is also passed. With `scheduleTask`, the task with the corresponding id can be started at the desired time. Here both tasks are started at the beginning.

```haskell
start :: Arduino ()
start = do
    setPinMode led OUTPUT
    setPinMode ledRed OUTPUT
    setPinMode digitalSoundPin INPUT

    createTask 1 $ ledTask
    createTask 2 $ soundTask
    scheduleTask 1 0
    scheduleTask 2 0
```

The translated C program will look as follows. First, `setup` is defined, which initiates the memory and starts the `haskinoMain` task. As can be seen, it is identical to that of the blink example.

```c
// simplified
void setup() {
    haskinoMemInit();
    // 255 is the id of the task haskinoMain
    createTask(255, haskinoMain);
    scheduleTask(255, 0);
    startScheduler();
}
```

The main program sets the pin modes as well as creating and scheduling the two tasks.

```c
  // simplified
  void haskinoMain() {
    pinMode(13, OUTPUT);
    pinMode(9, OUTPUT);
    pinMode(3, INPUT);

    createTask(1, task1);  // led task
    createTask(2, task2);  // sound task
    scheduleTask(1,0);
    scheduleTask(2,0);

    taskComplete();
  }
```

The first task with the blink program works as in the previous example.

```c
// simplified
void task1() { -- led task
    while (1) {
        digitalWrite(13,1);
        delayMilliseconds(1000);
        digitalWrite(13,0);
        delayMilliseconds(1000);
    }
    taskComplete();
}
```

The second task stores the value read by `digitalRead` under `bind3` and performs the corresponding `digitalWrite`.

```
// simplified
void task2() { -- sound task
    bool bind3;

    while (1) {
        bind3 = digitalRead(3);

        if (bind3) {
            digitalWrite(9,1);
        } else {
            digitalWrite(9,0);
        }

        delayMilliseconds(50);
    }
    taskComplete();
}
```

3 Using FRP on Arduino

The examples just described are reactive applications. The states of the LEDs change depending on the input or after the time specified by the delay has elapsed. Reactive applications can quickly become too complicated to program and therefore difficult to change. FRP is a declarative approach to programming reactive applications, which makes the code more modular. Furthermore, FRP provides a structured method for handling events that makes it easier to express complex event-driven behaviors.

By using FRP, the description of what the program is supposed to do is separated from the instructions to the output. In the blink example, the description of the program would be to change the state of an LED every second. The result can be represented as a time-continuous signal showing either zero or one and changing every second. The output sets the voltage level of the pin of the LED based on the current value of the signal. The output calls `digitalWrite` at regular intervals and passes the current value of the signal. So, the output does not care about how long an LED is supposed to stay on or off. If the delay is longer, the output calls `digitalWrite` more often with the same value.

This approach is more modular than the imperative approach. For example, it would be possible to replace the output to the hardware with an output to a graphical user interface (GUI) to simulate the execution of a program. Because of the modularization, the programming of the LED signal does not need to be adapted. So, it would be convenient to be able to use FRP when programming the Arduino as well. The following sections demonstrate how to implement the blink program for the Arduino in the shallow and deep EDSL using FRP.

3.1 FRP with Shallow EDSL

With Haskino's shallow EDSL, the use of FRP is not a problem because the program runs in the Haskell environment. This makes it possible to integrate external libraries into the program code. One FRP framework that has been used for robotics applications is Yampa. In Yampa, signals do not exist directly as types but can be created and transformed using signal functions. The type `SF a b` can be read as a signal function that takes a signal containing a value of type `a` as input and produces a signal containing a value of type `b` as output [HCNP03].

To implement the blink example using Yampa, a signal function is created that defines the behavior of the LED. This is then linked to the instructions for the LED pin. The behavior of the LED can be represented as a signal that indicates either `True` or `False`, depending on whether the LED should be turned on or off.

The blink program should first initialize the LED pin and then execute the program. In Yampa, this can be done with the `reactimate` function.

```
reactimate :: Monad m
  -- Initialization action
  => m a
  -- Input sensing action
  -> (Bool -> m (DTime, Maybe a))
  -- Output processing action
  -> (Bool -> b -> m Bool)
  -- Signal function
  -> SF a b
  -> m ()
```

The invocation of the function `reactimate` within the blink program looks like the following.

```
progBlink = do
    setPinMode led OUTPUT
    t <- liftIO $ getCurrentTime
    timeRef <- liftIO $ newIORef t
    reactimate
        firstInput
        \_ -> nextInputs timeRef
        \_ ledState -> output
        (toggle False)
```

The function `reactimate` runs `toggle` that defines the behavior of the LED. It generates a Boolean signal that toggles its value every second.

```
  toggle :: Bool -> SF a Bool
```

The resulting output signal is linked by `reactimate` to the desired output processing, in this case, the function `output`. This function executes the `digitalWrite` command to set the LED to the value received from the signal function. The discarded Boolean signifies a change in the output of `toggle`, while the return value indicates whether to stop the execution of `reactimate`.

```
output :: Bool -> Bool -> Arduino Bool
output _ x = do
  digitalWrite 13 x
  return False -- continue forever
```

The function `reactimate` also allows to pass external inputs to the signal function, e.g. when a button is pressed. These inputs can be passed in `nextInputs`. The argument indicates if it can block. In addition, an initial input can be specified in firstInput. For the blink example, no inputs are used.

```
firstInput :: Arduino ()   -- input at time zero
firstInput = return ()

nextInputs
   :: IORef UTCTime
   -> Bool
   -> Arduino (Double, Maybe ())
nextInputs timeRef _ = do
   now <- liftIO $ getCurrentTime
   lastTime <- liftIO $ readIORef timeRef
   liftIO $ writeIORef timeRef now
   let dt = now `diffUTCTime` lastTime
   return (realToFrac dt , Nothing)
```

3.2 FRP with Deep EDSL

Running an FRP program directly on the Arduino is more complex. The deep EDSL should represent the program to be executed as an AST. This means that any new functionality has to be integrated into the structure and the evaluation of the AST has to be adapted accordingly. In the case of Haskino, when translating from shallow EDSL to deep EDSL, the translation of the FRP functions in the example before is not defined. Therefore, Haskino has no way to run FRP directly on the Arduino at the moment.

However, it is possible to program the Arduino with FRP, as shown by the frp-arduino Haskell library. This deep EDSL allows writing FRP programs in Haskell and compiles them to C code that can be run on the Arduino. It is important to note that the frp-arduino library's capabilities are tailored specifically to the Arduino. In contrast, Yampa is an FRP library in Haskell that can produce a signal that can be interpreted by various systems.

The blink program can be implemented in frp-arduino as follows [Lin19].

```
digitalOutput pin13 =: clock ~> toggle
```

The clock function produces a signal (called a stream in the library) incrementing an integer at a given time interval. The ~> operator is used to pass the resulting signal to the toggle function. This produces a signal with one bit set based on the input signal. If the input signal contains an even number, the bit is set to 1, and if it contains an odd number, the bit is set to 0.

```
clock :: Stream Word

(~>) :: Stream a -> (Stream a -> Stream b) ->
    Stream b

toggle :: Stream Word -> Stream Bit
```

The =: operator appends the resulting output signal to pin 13. This turns the LED on and off depending on the signal.

```
(=:) :: Output a -> Stream a -> Action ()

digitalOutput :: GPIO -> Output Bit

pin13 :: GPIO
```

The Haskell program is then compiled into a C program and loaded onto the Arduino. The resulting C program can be seen in the examples of frp-arduino [Lin15]. To summarize the C program in a simplified way, a main loop is executed checking if a time interval has expired and, if so, executing digitalWrite with the corresponding bit.

It is therefore possible to take advantage of FRP also in deep EDSL. To use it in Haskino, the structure and the interpretation must be adapted accordingly.

4 Conclusion

Using Haskell instead of C to program the Arduino can make the program code more modular and therefore easier to program and maintain. There are several ways to run Haskell on embedded systems. A simple solution is a shallow EDSL, like the one provided by Haskino. The user can program Haskell as usual, using Haskell libraries such as Yampa. Running the code directly on the Arduino without a connection to a computer requires the use of a deep EDSL. This leads to a more efficient program execution, but also to a higher implementation effort. For example, when using a library like Yampa, the functionality has to be integrated into the deep EDSL.

The uses of EDSLs shown in this paper only allow the translation from Haskell to C code. In none of the variants can the hardware be controlled directly with Haskell. Furthermore, all the presented libraries are specifically designed for the Arduino. A more direct and general approach is provided by Conal Elliott [Ell17].

References

[Ard23] Arduino. Arduino documentation: Blink. https://docs.arduino.cc/built-in-examples/basics/Blink, 2023. Accessed: 2023-05-15.

[Aug12] Lennart Augustsson. Tech mesh 2012 - making edsls fly - lennart augustsson. https://www.youtube.com/watch?v=7gF7iFB4mFY, 2012. Accessed: 2023-05-15.

[Baa09] Christiaan Baaij. Cλash : from haskell to hardware, December 2009.

[Ell17] Conal Elliott. Compiling to categories. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.

[EPW15] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. Guilt free ivory. In *Haskell Symposium*. ACM, 2015. Available at https://leepike.github.io/pub_pages/ivory.html.

[Erk22] Levent Erkök. Hackage package: harduino. https://hackage.haskell.org/package/hArduino, 2022. Accessed: 2023-05-15.

[Fow19] Martin Fowler. Domain-specific languages guide. https://martinfowler.com/dsl.html, 2019. Accessed: 2023-06-14.

[GD16] Andy Gill and Justin Dawson. Hackage package: remote-monad. https://hackage.haskell.org/package/remote-monad, 2016. Accessed: 2023-06-14.

[GG19] Mark Grebe and Andy Gill. Threading the arduino with haskell. In David Van Horn and John Hughes, editors, *Trends in Functional Programming*, pages 135–154, Cham, 2019. Springer International Publishing.

[GSD15] Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. The remote monad design pattern. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, page 59–70, New York, NY, USA, 2015. Association for Computing Machinery.

[GYG17] Mark Grebe, David Young, and Andy Gill. Rewriting a shallow dsl using a ghc compiler extension. *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2017.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Lin15] Rickard Lindberg. frp-arduino examples: Blink.c. https://github.com/frp-arduino/frp-arduino/blob/master/examples/Blink.c, 2015.

[Lin19] Rickard Lindberg. frp-arduino. https://github.com/frp-arduino/frp-arduino, 2019.

[PWNG12] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: A do-it-yourself high-assurance compiler. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 335–340, New York, NY, USA, 2012. Association for Computing Machinery.