Exploring the use of Haskell to Program Microcontrollers used in Educational Robotics Platforms

Olivier Lischer
OST Eastern Switzerland University of Applied Sciences
MSE semester project 1
Supervisor: Farhad Mehta
Autumn 2024

Abstract

Nowadays children are introduced to electronics and programming by developing applications for "educational robotic kits". These kits allow children to write simple applications for a robot that interacts with its environment. Because Python or a block based system like Scratch are beginner friendly, they are often supported by these kits. Sometimes event C/C++ SDKs are available for more advanced users. None of the common kits support a functional programming language and therefore the children do not have a chance to try a different approach to programming. Previous research has proved that it is possible to write a Haskell application on bare metal by implementing an operating system. With MicroHs, a newer Haskell compiler is developed, that is based on combinators. The author of MicroHs has already demonstrated that it is possible to write simple MiroHs applications that run on micro controllers. A different approach is used in the project Categorifier. It utilizes the "Compiling to categories" to transform Haskell code into C code. In this project, MicroHs is used to develop a line following algorithm that runs on a Raspberry Pi Pico. The Raspberry Pi Pico controls a PicoGo robot to demonstrate, how a functional programming language can be used for an educational robotic kit. While it is possible to develop Haskell applications for devices with little memory, it is rather difficult as the Haskell runtime must be adjusted and a lot of C code is still required to get it on to the device.

Keywords: Haskell, education, embedded, robotics

1 Introduction

To introduce children in schools to electronics and programming, "educational robotic kits" are being developed. Traditionally, systems programming languages such as C/C++ are used for embedded development and controlling hardware. To simplify the process and allow children to focus on the general concepts, a Scratch-based IDE is often provided alongside the kit. MicroPython is also often offered as an option for older children and more advanced use cases.

These technologies all follow the imperative programming paradigm. Children using these technologies are introduced to thinking in an imperative model from an early stage. Currently, there are no kits available that are targeted at functional programming languages. This project demonstrates how a beginner-friendly kit can be programmed using Haskell, a functional programming language.

In section 2, a review of existing educational robotic platforms is performed while in the following section 3, more details are given for the Raspberry Pi Pico as it is the chosen hardware for this project. Furthermore, possible technologies to run Haskell are introduced.

In section 4, a complete example of how to develop an application using Haskell for the Raspberry Pi Pico is shown.

2 Review of the current state-of-the-art in educational robotic platforms

In this section, various educational robotic platforms are compared. The kits are compared regarding the following criteria:

- Underlying chip
- Sensors and actuators
- Available programming languages for the kit
- Licensing for hardware and software

2.1 Thymio

Thymio is an educational robot designed by researchers form the EPFL and produced by Mobsya. Thymio is powered by the Aseba VM[Mob24].

The primary IDE for developing applications is Aseba Studio. It supports VPL and Scratch; both graphical ways to develop applications and Aseba the main language for Thymio[Pro]. Aseba Studio also communicates with the so-called Thymio Device Manager (TDM) that compiles the source to byte code and sends it to the robot.

Furthermore, it is possible to use Python using the Python package `tdmclient` to develop applications for Thymio[Tdm]. When there is no option to compile to Aseba or one wants to communicate with Thymio directly without communicating over the TDM, the Python package `thymiodirect` could be used[Thy]. This Python package allows to send Aseba byte code directly to Thymio. This way, every programming language can be used to control the Thymio, as long as it is possible to compile it to Aseba bytecode. In

Figure 1, the interactions between the previously described parts are graphically recorded.

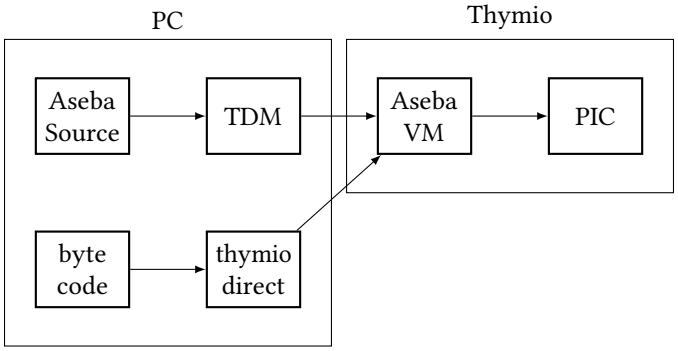

Figure 1. The Thymio architecture

The robot is based on the PIC24FJ128GB106-I/PT micro controller. It is further equipped with the following sensors:

- IR proximity sensors
- capacitive touch buttons
- three-axis accelerometer
- microphone (recording or noise detection)
- IR receiver (for remote control)
- wireless module

Additionally, it has three types of actuators: 39 LEDs, two DC motors and one speaker. To communicate with other devices, Wi-Fi can be used[Bon].

Thymio's hardware design is in the public domain under Creative Commons while the software stack is licensed under LGPL-3.0[Bon, Mob24].

2.2 micro:bit

The *micro:bit* is a programmable device by BBC and targets children from the age of eight years or older[Wha]. It is based on the Arm Cortex M4 32bit and is equipped with a motion and a temperature sensing sensor. Further, the micro:bit is capable of communicating with its environment using Bluetooth and low level radio communication. It also provides further features[New]:

- nRF52 Application Processor (where the user application is run)
- 2 user buttons, 1 system button (reset)
- Display (5×5 array of LEDs)
- Speaker (JIANGSU HUANENG MLT-8530)
- Microphone (Knowles SPU0410LR5H-QB-7 MEMS)
- 19 GPIOs

The micro:bit's architecture is publicly documented under https://tech.microbit.org/hardware/.

The micro:bit can be programmed using *Microsoft MakeCode*, a block based IDE. More advanced users can use Python or the C/C++. Micro:bit provides a web editor including a simulator under https://python.microbit.org/v/3, where the application can be tested and debugged before transferring it onto the micro:bit hardware[Mic].

2.3 Makeblock mBot2

The *Makeblock mBot2* is a platform based on the CyperPi, which is produced by the Chinese company Makeblock, where the actuators are already connected. The CyperPi is the processing unit that controls how the mBot2 as a whole interacts with its environment. The CyberPi is based on the ESP32-WROVER-B and provides a light sensor as well as a gyroscope. To communicate with its environment, Wi-Fi and dual-mode Bluetooth are available[Wha22].

The CyperPi is programmed using mBlock 5, a block based IDE or using Python. The developers of the CyperPi are providing a Python library called *cyperpi* to work with Python. A web based version of mBlock 5 is available under https://ide.mblock.cc/. There one can switch between block based programming and Python[Pro22].

2.4 Lego Education Spike

The *Lego Education Spike* is the successor of the Lego Mindstorms series that is discontinued. The larger system is based on the STM32F413 (Architecture: ARM Cortex M4) and has a gyro sensor included. The system supports Bluetooth for wireless communication. It also has:

- 5×5 LED matrix
- input / output ports
- buttons
- speaker

The Technic Large Hub can be programmed using a block based method as well as with Python. For this, the web editor is provided under https://spike.legoeducation.com/[LEG].

2.5 Arduino Alvik

The *Arduino Alvik* is based on the Arduino Nano ESP32 microcontroller. The set can communicate using Wi-Fi and Bluetooth LE. Moreover, the kit is equipped with multiple sensors:

- RGB color detection
- IMU
- time of Flight distance sensor
- line follower
- capacitive touch sensor

The Arduino Nano ESP32 is open source[Lic24]. Besides MicroPython, languages such as C/C++ are also available to program the micro controller[Ard].

2.6 Raspberry Pi Pico

The *Raspberry Pi Pico* is as a micro controller targeting beginners and advanced developers. There are two versions of the Raspberry Pi Pico:

- RP2040
- RP2350 (Pico 2)

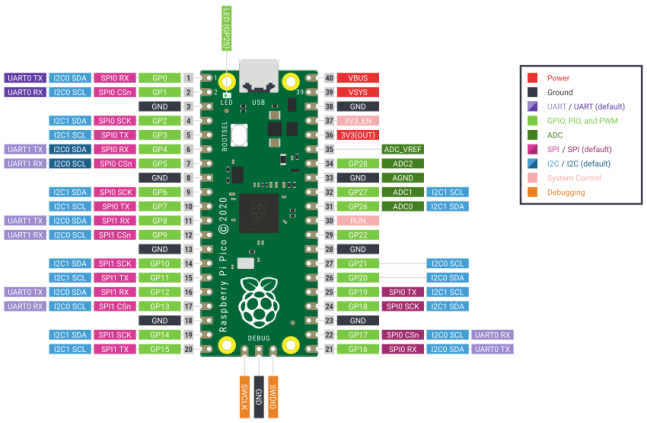

Figure 2. Pinout RP2040

The Raspberry Pi Pico series comes with a C/C++ SDK, that provides a friendly API to use the features from the hardware. Additionally, to C/C++ there is MicroPython available a well[Ras].

The RP2040, the first version of the pico, is based on the Arm Cortex M0+ and has 264KB of SRAM and 2MB on-board flash memory. The Raspberry Pi Pico only has a temperature sensor built-in, but it comes with 26 multi-function GPIO pins where various actuators and sensors can be connected to. The exact pinout can be seen in Figure 2. To flash the application on the Raspberry Pi Pico, a simple Drag-and-drop system is available. This makes it attractive for beginners as well as for experienced developers[Pic].

The second version of the Pico family, the RP2350, is an upgrade to version 1. The Raspberry Pi Pico 2 is based on the Cortex-M33 or Hazard3. With 520KB of SRAM and 4MB of on-board flash memory, the Raspberry Pi Pico 2 has twice as much memory as version 1. The only sensor is a temperature sensor[Pic].

There are no starter or educational kits available for the Raspberry Pi Pic. However, there are prebuilt sets of sensors and actuators that can be used to develop simple to advanced applications without tinkering with the hardware.

2.7 Honorable mentions

There are also some RISC based microcontrollers suitable for education. BeagleBoard is a company developing such devices with the *BeagleV-Ahead* and *BeagleV-Fire*[Boa]. There exists the *PocketBeagle Gorve Kit* with which students can learn the basics of embedded programming[Poc].

2.8 Summary review

In Table 1 all kits and microcontrollers from the previous sections are recorded and compared in licensing model, the languages that can be used to control it and if it is in general an open source (OS) project. It can be seen that Python can be used for all platforms. Therefore, if a new programming language should be used on educational kits, one could provide a transpiler to Python and gain the possibility to use the language on many platforms without implementing specific support for them. For this project, the Raspberry Pi Picos 1 and 2 were chosen because it is open source and is suitable for beginners. There is a large selection of sensors and actuators as well as ready-made sets that can be used together with the RP2040 and RP2350.

Kit name	OS	Languages
Thymio	YES	Block, Python, Aseba
Micro:bit	YES	C/C++, Block, Python
Makeblock mBot2	NO	Block, Python
Lego Education Spike	NO	Block, Python
Arduino Alvik	YES	C/C++, Python
RP2040	YES	C/C++, Python
RP2350	YES	C/C++, Python
beagleboard	YES	Python

Table 1. Summary over the kits

3 Haskell for the Raspberry Pi Pico

It was shown in section 2 that some kind of block based programming, Python and C/C++ are very common to develop application for educational kits. None of them support Haskell by default or any other functional programming language. In this section, possible technologies are presented how Haskell code can be compiled for educational kits. As a concrete example, the Raspberry Pi Pico Version 1 (RP2040) is used.

3.1 Compile Haskell

Haskell code must first be compiled to a suitable binary format before it can be used for any micro controller. This task is executed by the compiler. At the time of writing, two Haskell compilers can be considered as actively maintained:

- GHC (de facto standard compiler for Haskell)
- MicroHs

In addition to a compiler that compiles to a binary file, there is the idea of "Compiling to categories" by Conal Elliott [Ell17]. The project Categorifier is a GHC compiler plugin that is a real-world implementation of this idea. This idea is discussed in detail in subsection 3.4.

3.2 GHC

GHC is the de facto standard compiler for modern Haskell code. By developing the Haskell operating system "House", it was proved that it is possible to develop applications for bare metal platforms using GHC. The authors achieved the compilation of Haskell code to bare metal by porting GHC to the IA-32 architecture[HJLT05]. This is not feasible for this

project as micro controllers usually do not have the required amount of memory to run the GHC runtime and the user application.

3.3 Introduction to MicroHs

MicroHs is an extended subset of Haskell that uses combinators for the runtime execution[Aug24]. The combinators and the evaluation machinery are based on the paper by D. A. Turner[Tur79] but are extended with custom ones.

The MicroHs compiler can be instructed to compile the following targets from Haskell source code:

- combinators
- C file with the combinators stored in an array
- regular executable including the runtime system (RTS)

The first option can be evaluated by any RTS as long as the RTS supports all combinators. This circumstance can be used to support any device by porting the current MicroHs RTS written in C to a suitable language. For this project, the second option was chosen as it provides the most flexibility without the need to develop a new RTS.

MicroHs requires a configuration in the form of a C header file so that a binary can be created for the target platform. In that configuration files various options can be adjusted. A selection of possible options are:

- IO functions
- float operations
- heap and stack size

In this configuration initialization, clean up and custom C functions can be provided as well. In Listing 1 a minimal working configuration for the Raspberry Pi Pico is shown. It explicitly disables IO and performs an initialization.

```
#ifndef CONFIG_RASPERRYPICO_H
#define CONFIG_RASPERRYPICO_H

#define WANT_STDIO 0
#define WANT_MATH 0
#define WANT_MD5 0
#define WANT_TICK 0
#define WANT_ARGS 0
#define GCRED 0
#define FASTTAGS 0
#define INTTABLE 0
#define SANITY 0
#define STACKOVL 0

#define HEAP_CELLS 4000
#define STACK_SIZE 500

#include "pico/stdlib.h"
#include <stdio.h>

void pico_set_led(bool led_on) { gpio_put(
    PICO_DEFAULT_LED_PIN, led_on); }

#define INITIALIZATION
void main_setup(void) {
  stdio_init_all();
  gpio_init(PICO_DEFAULT_LED_PIN);
  gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
  for (int i = 0; i < 10; i++) {
    pico_set_led(true);
    sleep_ms(100);
    pico_set_led(false);
    sleep_ms(100);
  }
}

void myexit(int n) {
  while (true) {
    pico_set_led(true);
    sleep_ms(250);
    pico_set_led(false);
    sleep_ms(250);
  }
}
#define EXIT myexit

#endif /* CONFIG_RASPERRYPICO_H */
```

Listing 1. Snippet from config-raspberry-pico.h

In addition to the configuration file, a normal C file is required. The C file combines the evaluation of the runtime system (`eval.c`) and the configuration for the target platform.

```
#include "config-raspberry-pico.h"
#include "eval.c"
```

Listing 2. eval-raspberry-pico.c

With that in place, MicroHs code can be compiled for the target platform.

3.4 Compiling to Categories

Conal Elliott presented in his paper "Compiling to Categories" that Haskell code can be compiled to any other Cartesian Closed Category (CCC) using GHC[Ell17]. A GHC compiler plugin has been developed to enable this. This compiler plugin takes the GHC Core language as input and performs the transformation to the target category, such as C code, at compile time. Embedded Domain-Specific Languages (EDSL) can do the same. However, EDSL often perform the same work as the compiler at runtime as a library. The advantage over EDSL is that the transformation is done at compile time by the compiler, not at run-time by a library. This has the advantage of maintaining less code and often leads to better optimized output, since the compiler is optimized for such cases.

This approach was used by the company Kittyhawk to control their aircraft. They extended the original plugin and developed the GHC compiler plugin Categorifier[Pfe24].

3.5 Development of a proof-of-concepts

Before developing an advanced application using MicroHs and the Raspberry Pi Pico, a proof-of-concepts was developed to test if and how a working application could be developed. That included to write a small application that reads and writes data over GPIO. In the following, the build process for MicroHs applications on a Raspberry Pi Pico is described.

First, the Haskell code is compiled to a combinators representation. This representation is stored inside a C array.

When using Foreign Function Interface (FFI), the MicroHs compiler will generate the required glue code, so that the runtime can perform the required conversion between C and Haskell, and stores it next to the combinators. The FFI mechanism will be used, to call C functions from the Raspberry Pi Pico SDK. If applications require user-defined C functions that should be callable from the Haskell application, writing them in an additional header file is the intended way. The MicroHs compiler will include the required header file automatically into the C combinators file. A reduce example of such a combinators file can be seen in Listing 3.

```c
static unsigned char data[] = {/* Compiled
    combinators */};
unsigned char *combexpr = data;
int combexprlen = 637;
#include "mhsffi.h"
#include "pico/stdlib.h"
void mhs_gpio_get(int s) { mhs_from_Int(s, 1,
    gpio_get(mhs_to_Int(s, 0))); }
static struct ffi_entry table[] = {
{ "gpio_get", mhs_gpio_get},
{ 0,0 }
};
struct ffi_entry *xffi_table = table;
```

Listing 3. blinky-comb.c

The building process is summarized in Figure 3.

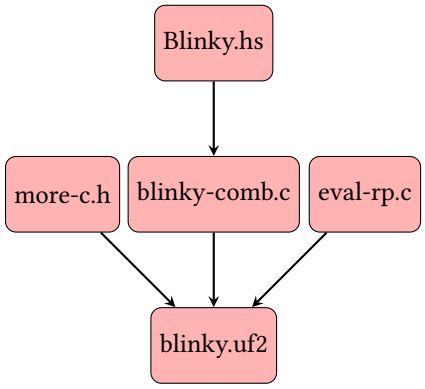

Figure 3. The building process using MicroHs for a Raspberry Pi Pico

In Listing 4 the Haskell code of the first applications is listed. It turns the default built-in LED on and off as well as an external LED that is connected via a GPIO pin. The full source code for the proof-of-concepts can be found on GitHub.

```haskell
module Blinky(main) where
import Prelude

defaultLed :: Int
defaultLed = 25

gpioLed :: Int
gpioLed = 20

gpioOut :: Int
gpioOut = 1

main :: IO ()
main = init >> blinky

init :: IO ()
init = do
  c_gpio_init gpioLed
  c_gpio_init defaultLed
  c_gpio_set_dir gpioLed gpioOut
  c_gpio_set_dir defaultLed gpioOut

blinky :: IO ()
blinky = do
  setLed True
  wait 250
  setLed False
  wait 250
  blinky

setLed :: Bool -> IO ()
setLed on = do
  c_gpio_put defaultLed $ if on then 1 else 0
  c_gpio_put gpioLed $ if on then 1 else 0

wait :: Int -> IO ()
wait = c_sleep_ms

-- C imports omitted
```

Listing 4. Blinky.hs

As can be seen, it is possible to directly use the C Raspberry Pi Pico SDK using Haskell's Foreign Function Interface capabilities. However, to achieve better integration into the Haskell language, a wrapper around the C function should be written.

4 Line-following with MicroHs on the PicoGo Robot

In this section, the process of developing a line-following robot is described. The PicoGo Robot is used for the hardware. The full source code of this application can be found at https://gitlab.ost.ch/robohs/picohs.

4.1 Streamline the development process

Before starting to develop the application for the PicoGo Robot, a cmake build script was developed to simplify the compilation from MicroHs to a binary suitable for the Raspberry Pi Pico. With the cmake approach, there is no manual building of single compilation steps required, but only two:

- generating building scripts using cmake
- compiling using cmake

To enable faster developing iterations, software is often used to simulate the target platform. Such a simulator allows developing the application without copying the binary after every change to the target platform. For the Raspberry Pi Pico there is the online simulator wokwi available. As MicroHs is not standard way to build binaries for these platforms, one had to upload the compiled binary to run it inside the simulator. However, the platform is not suitable for this

project, as not all required hardware parts are available to reproduce the desired hardware behaviour. Therefore, testing must be done by coping the binary on to the hardware.

4.2 Powering the motors

The motors of the PicoGo robot are connected to the micro controller over GPIO pins. These GPIO pins must be initialised and configured, as well as the PWM blocks. To hide these hardware interactions and to save as much memory as possible, this was done directly in C. The motor state is kept in the C file as global variables. The operations to control the motors were implemented in C as well and then exported to Haskell.

To prevent Haskell users to use the driving functions before the motor is initialised, the `Motor` module (Listing 5) exposes only a smart constructor and wrappers around the C functions that requires an already constructed motor. The Haskell user must call `init` and is then able to drive the PicoGo as desired.

```
module Motor
  ( Motor
  , init
  , forward
  , backward
  , left
  , right
  , stop
  , set
  ) where
import Prelude

data Motor = Motor

init :: IO Motor
init = c_motor_init >> return Motor

forward  :: Motor -> Int -> IO ()
forward _ = c_motor_forward

backward :: Motor -> Int -> IO ()
backward _ = c_motor_backward

left :: Motor -> Int -> IO ()
left _ = c_motor_left

right :: Motor -> Int -> IO ()
right _ = c_motor_right

stop _ = c_motor_stop

type LeftSpeed  = Int
type RightSpeed = Int

set :: Motor -> LeftSpeed -> RightSpeed -> IO ()
set _ = c_motor_set

-- C imports omitted
```

Listing 5. Motor.hs

4.3 Reading the sensors

Similar to the motor, the 5 IR sensors are read through GPIO pins. Using Programmable Input/Output (PIO) an SPI interface between the IR sensors and the Raspberry Pi Pico was implemented. The initialization of the PIO application is done using C macros. As C macros are evaluated only at the time of compiling, the initialisation must be done in C regardless how much memory is available.

The Haskell module `Sensor` in Listing 6 provides a smart constructor to initialize the sensors similar to the `Motor` module. The module also provides the `readLine` function to read the sensors and get calibrated values. As C and Haskell can not exchange arrays, but only pointers and certain primitives, memory must be allocated upfront. It was decided to allocate the memory during sensor initialization on the C side and passing a pointer to Haskell when calling the C function `c_sensor_read_line`. This has the benefit, that the memory must be allocated only once at initialization and no further over head of memory allocation must be paid. However, care must be taken to not release the allocated memory for the lifetime of the application or to not read over the allocated memory buffer.

```
module Sensor
  ( Sensor
  , LineColor(..)
  , init
  , readLine
  ) where
import Prelude
import Foreign
import Control.Monad (mapM)

data Sensor = Sensor
data LineColor = Black | White

init :: IO Sensor
init = c_sensor_init >> return Sensor

readLine :: Sensor -> LineColor -> IO (Int, [Int])
readLine _ color = do
  ptr <- c_sensor_read_line (isWhiteLine color) >>=
       newForeignPtr_
  xs <- withForeignPtr ptr getReadLineValuesWithPtr
  return (head xs, tail xs)

isWhiteLine :: LineColor -> Int
isWhiteLine White = 1
isWhiteLine Black = 0

getReadLineValuesWithPtr :: Ptr Word16 -> IO [Int]
getReadLineValuesWithPtr ptr = mapM (
     c_get_value_with_ptr ptr) [0..5]

-- C imports omitted
```

Listing 6. Sensor.hs

4.4 Adjusting heap and stack size

The default values for heap and stack size in MicroHs are not suitable for an embedded device as it targets modern

computers with a lot of memory. To be able to run a MicroHs application on a micro controller, these values must be adjusted. To calculate the upper bound for the maximum number of heap cells Equation 1 can be used.

$$\text{Heap}_{max} = \frac{\text{Memory}_{tot}}{\text{size of node}} \tag{1}$$

For the Raspberry Pi Pico RP2040 this results in Equation 2

$$\begin{aligned} \text{Heap}_{max} &= \frac{\text{Memory}_{tot}}{\text{size of node}} \\ &= \frac{264KB}{16B} \\ &= 165000 \end{aligned} \tag{2}$$

The resulting number is the upper bound. The value must be gradually decreased from there. A working number of heap cells was at 13000 and stack size at 500.

4.5 Line following algorithm

The algorithm itself can be written in pure Haskell. It calculates the power for each motor (left and right) so that the car is moving along the line. An example for such an algorithm can be found in Listing 7.

```haskell
module Drive(main) where
-- Imports and constants omitted

data Car = Car Motor Sensor
type State = (Int, Int)

main :: IO ()
main = initCar >>= flip appLoop (0, 0)

initCar :: IO Car
initCar = do
  motor <- M.init
  sensor <- S.init
  return $ Car motor sensor

appLoop :: Car -> State -> IO ()
appLoop car@(Car motor sensor) st = do
  (position, _) <- S.readLine sensor Black
  let (pd, st') = update st position
  uncurry (M.set motor) (calcMotorConfig pd)
  appLoop car st'

calcMotorConfig :: Int -> (Int, Int)
calcMotorConfig pd = if pd < 0 then (maxSpeed + pd,
     maxSpeed) else (maxSpeed, maxSpeed - pd)

update :: State -> Int -> (Int, State)
update (prop, int) pos = (pd, (p, i))
  where
    p  = calcProportional pos
    d  = calcDerivative prop p
    i  = calcIntegral int p
    pd = calcPowerDifference p d i

calcProportional :: Int -> Int
calcProportional position = position - 3000

calcDerivative :: Int -> Int -> Int
calcDerivative x0 x1 = x1 - x0

calcIntegral :: Int -> Int -> Int
calcIntegral x0 v = clamp (-5000, 5000) x0 + v

calcPowerDifference :: Int -> Int -> Int -> Int
calcPowerDifference proportional derivative integral
      = clamp (-maxSpeed, maxSpeed) $ prop + der +
     int
  where
    toInt = fromInteger . truncate
    prop = toInt $ fromIntegral proportional * p
    der = toInt $ fromIntegral derivative * d
    int  = toInt $ fromIntegral integral * i
```

Listing 7. Drive.hs

5 Further work

In this project, MicroHs was used to develop an application for a micro controller. Currently, the way to develop a Haskell application for an embedded device using MicroHs is not yet suitable for children. First of all, a student has to write C code to interact with the hardware and they have to think about how to handle memory. Furthermore, manual adjustments to the runtime system are required in order to enable the execution of the application on low-memory devices. That could be improved by implementing MicroHs configurations that provide all the necessary C functions to interact with the hardware and wrapping them using a Haskell FFI wrapper. Moreover, MicroHs is still a young project with a relatively small user base. So it is possible, that there are smaller bugs in the MicroHs compiler or runtime system. This can be improved by contributing to the project by using it, creating issues or submitting pull requests.

An alternative approach would be the idea “Compiling to Categories” using the GHC plugins “concat” and “categorifier”. These plugins are able to generate appropriate C code. A “Raspberry Pi Pico Category” must be prepared in advance, so that the children can concentrate on writing applications for their device.

6 Conclusion

As it was shown in section 4 was demonstrated, it is possible to control a robot using Haskell. However, interacting with the hardware still requires a significant amount of C code. Nonetheless, one could write the business logic in Haskell while limiting the hardware interactions to C.

Writing Haskell programs for the Raspberry Pi Pico is difficult because the tooling around MicroHs is not well developed yet. To make matters worse, the current process is to compile to C and from there to a binary. This adds an abstraction layer and makes debugging more difficult.

While, there is a simulator available for the Raspberry Pi Pico, the selection of available hardware components was not sufficient for this project. Therefore, the binary muse be regularly copied to the board for testing. This adds additional developing overhead and requires that the physical components are available and attached to the micro controller.

With regard to teaching children the fundamentals of functional programming, this method is not yet sufficiently developed. Developing an application using MicroHs for the Raspberry Pi Pico requires C/C++ knowledge and requires manual adjustments to the MicroHs runtime.

References

[Ard] Arduino Alvik. https://www.arduino.cc/education/arduino-alvik/.

[Aug24] Lennart Augustsson. MicroHs: A Small Compiler for Haskell. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium*, pages 120–124, Milan Italy, August 2024. ACM.

[Boa] Board Selection. https://www.beagleboard.org/boards.

[Bon] Michael Bonani. 1 Laboratoire de Systèmes Robootiques.

[Ell17] Conal Elliott. Compiling to categories. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–27, August 2017.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 116–128, New York, NY, USA, September 2005. Association for Computing Machinery.

[LEG] LEGO® Education SPIKE™ Prime Technical Specifications. https://assets.education.lego.com/v3/assets/blt293eea581807678a/bltf512a371e82f6420/5f8801baf4f4cf0fa39d2feb/techspecs_techniclargehub.pdf?locale=en-us.

[Lic24] Licensing for products based on Arduino. https://support.arduino.cc/hc/en-us/articles/4415094490770-Licensing-for-products-based-on-Arduino, July 2024.

[Mic] The micro:bit runtime DAL/CODAL. https://tech.microbit.org/software/runtime/.

[Mob24] Mobsya/aseba. https://github.com/Mobsya/aseba, August 2024.

[New] The new micro:bit V2. https://support.microbit.org/support/solutions/articles/1900 details-of-micro-bit-v2.

[Pfe24] Greg Pfeil. Sellout/compiling-anything-to-categories. https://github.com/sellout/compiling-anything-to-categories, February 2024.

[Pic] Pico-series Microcontrollers - Raspberry Pi Documentation. https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html.

[Poc] PocketBeagle® Grove Kit. https://www.beagleboard.org/boards/pocketbeagle-grove-kit.

[Pro] Programming with Thymio Suite. https://www.thymio.org/products/programming-with-thymio-suite/.

[Pro22] Programming Software. https://support.makeblock.com/hc/en-us/articles/7048529365271-Programming-Software, June 2022.

[Ras] Raspberry Pi Documentation - Microcontrollers. https://www.raspberrypi.com/documentation/microcontrollers/.

[Tdm] Tdmclient: Communication with Thymio II robot via the Thymio Device Manager. https://github.com/epfl-mobots/tdm-python.

[Thy] Thymiodirect: Communication with Thymio II robot via serial port or TCP. https://github.com/epfl-mobots/thymio-python.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, January 1979.

[Wha] What is the micro:bit? https://microbit.org/get-started/what-is-the-microbit/.

[Wha22] What is CyberPi. https://support.makeblock.com/hc/en-us/articles/7047875941399-What-is-CyberPi, June 2022.