Domain Specific Languages for Small Embedded Systems

By

Mark Grebe

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Committee members

Dr. Andy Gill, Chairperson

Dr. Perry Alexander

Dr. Prasad Kulkarni

Dr. Suzanne Shontz

Dr. Kyle Camarda

Date defended: _____

The Dissertation Committee for Mark Grebe certifies
that this is the approved version of the following dissertation :

Domain Specific Languages for Small Embedded Systems

Dr. Andy Gill, Chairperson

Date approved: _____

Abstract

Resource limited embedded systems provide a great challenge to programming using functional languages. Although these embedded systems cannot be programmed directly with Haskell, I show that an embedded domain specific language is able to be used to program them, and provides a user friendly environment for both prototyping and full development. The Arduino line of microcontroller boards provide a versatile, low cost and popular platform for development of these resource limited systems, and I use these boards as the platform for my DSL research.

First, I provide a shallowly embedded domain specific language, and a firmware interpreter, allowing the user to program the Arduino while tethered to a host computer. Shallow EDSLs allow a programmer to program using many of the features of a host language and its syntax, but sacrifice performance. Next, I add a deeply embedded version, allowing the interpreter to run standalone from the host computer, as well as allowing the code to be compiled to C and then machine code for efficient operation. Deep EDSLs provide better performance and flexibility, through the ability to manipulate the abstract syntax tree of the DSL program, but sacrifice syntactical similarity to the host language. Using Haskino, my EDSL designed for Arduino microcontrollers, and a compiler plugin for the Haskell GHC compiler, I show a method for combining the best aspects of shallow and deep EDSLs. The programmer is able to write in the shallow EDSL, and have it automatically transformed into the deep EDSL. This allows the EDSL user to benefit from powerful aspects of the host language, Haskell, while meeting the demanding resource constraints of the small embedded processing environment.

Acknowledgements

I am indebted to a great number of people for helping me during my graduate school journey of the last 8 years. This document is the result of support and guidance that I have received from many along the way.

First and foremost, I would like to thank my advisor, Andy Gill, for providing excellent guidance, support and mentoring. I learned an enormous amount from him, and appreciate all of his help introducing me to aspects of the academic world.

To all of my mentors and peers at the ITTC lab, I have enjoyed working with everyone, and learned a lot from all of you as well. Thanks Perry, Prasad, Suzanne, Garrett, Ed, Andrew, Justin, Jason, Mike, Adam, Paul, David, Forrest, and Tyler.

Many thanks also go to my Master's advisor, Kenneth Anderson, at the University of Colorado, as well as to John Boye and Khalid Sayood at the University of Nebraska, who's advice and mentorship have been invaluable throughout my career.

I would like to thank the two managers I worked for during my time in graduate school, Seth Cramer and Ted Mabie. Without their support, encouragement, and flexibility, it would have been a challenging task to get to this point.

And finally to the two people who have inspired me the most, and who deserve all the thanks in the world for enduring the many trials and stresses of supporting me while I worked full time and attended grad school, my wife Kim, and my daughter Tara. There are not enough words in the world to thank you properly, so these few here will have to do, but I love you both more than I can say.

This material is based upon work supported by the National Science Foundation under Grant No. 1350901. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Small, resource constrained embedded systems provide a challenge to programming with high level functional languages. Their small RAM and permanent storage resources make it impossible to run languages like Haskell directly on them. An alternative to using a high level language directly on such systems, is to use an Embedded Domain Specific Language (EDSL). Using an EDSL allows the user to write code using a subset of the look, feel, and semantics of the host language.

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

1.1 Embedded Domain Specific Languages

Embedded domain specific languages come in two flavors, shallowly embedded and deeply embedded. Shallowly embedded DSLs compute values directly, while deeply embedded DSLs build an abstract syntax tree of computations. With shallow EDSLs, values are computed directly, and chaining together computations requires the involvement of the host language. The syntax and semantics of a shallow EDSL are much closer to the syntax and semantics of a host language. The result of a computation in a deep EDSL is a structure, which may be used to cross-compile the computation before being evaluated. This ability does come at a cost, as deep EDSL's often require special syntactic notations for language features such as control structures.

Table 1.1: EDSL Options

	Native Execution/ Interpretation	**Code Generation/ Compilation**
Shallow EDSL	*Examples* hArduino Blank Canvas Haxl *Advantages* Ease of development Quick turnaround	*Examples* Haskino *Advantages* Ease of development Performance Resource Optimization
Deep EDSL	 *Advantages* Deep Debugging	*Examples* Kansas Lava Feldspar Ivory *Advantages* Performance Resource Optimization

Table 1.1 illustrates a subset of the EDSLs hosted by Haskell, my EDSL's host language. The vertical axes is divided into shallowly and deeply embedded EDSL's. The horizontal axis describes the method of executing the computation on the target system. EDSL's in the first column either

execute the computation natively in the host language, or package the computation for execution in another form, such as a bytecode, for interpretation either locally or remotely. The systems in the second column use the host representation of the computation to generate code in another language, and compile that language for execution on the target system. One way of packaging the computation for remote interpretation, as is done by systems in the first column, and which is used in much of our research, is to use the Remote Monad (Gill et al., 2015) design pattern. This design pattern allows for bundling of computations to be sent to a remote target.

Shown in the upper left quadrant of the table are EDSLs which share the attributes of being shallowly embedded DSL's, implemented using the Remote Monad design pattern. hArduino (Erkok, 2014) is an EDSL used to program small embedded systems based on the Arduino series of boards. It is shallowly embedded, and may only be used while tethered to a host with a USB cable. It is the predecessor to the Haskell Arduino system I have developed. Blank Canvas (Gill & Scott, 2015) is a shallowly embedded EDSL which allows users to program interactive images in a web browser using the HTML5 Canvas API in Haskell. Haxl (Marlow et al., 2014) is a Haskell library and EDSL for efficient access of remote data sources, and is also shallowly embedded. These EDSLs share the characteristic of ease of development, since given their shallow embedding, their syntax is close to idiomatic Haskell. They also allow quick turnaround, as intermediate results of computations can be observed on the host, allowing ease of debugging.

The lower right quadrant of the table lists examples of EDSL's which are deeply embedded and include code generation. Kansas Lava (Gill et al., 2013) is an EDSL for hardware entities, and is able to generate VHDL. Felsdspar (Axelsson et al., 2010)(Axelsson et al., 2011) is an EDSL for describing digital signal processing algorithms, and is able to generate C language code from the EDSL. Ivory (Elliott et al., 2015) is an EDSL that is designed as a language for safe systems level programming, and also generates C language code. All of these EDSL's have the characteristics of better performance and better resource utilization than shallow EDSL's, due to the ability to generate code in a low level language.

The lower left quadrant in the table is the worst of both worlds. With EDSLs in this quadrant, the user must write in a harder to use deep language, and live with lower performance and suboptimal resource utilization, since code is not generated in a lower level language. One use we have found for languages in this quadrant is debugging the development of a Deep EDSL prior to code generation.

The upper right quadrant is where systems that combine ease of use and optimized performance may be found. EDSLs in this quadrant are able to be written in the easier to use shallow embedding, and have the better performance and resource utilization characteristics of code generation. Also, if the same shallow code can be used with native execution/interpretation, as well as code generation, the user can first prototype with native execution or interpretation, and then deploy with code generation. It is a solution that enables programming in this quadrant that I present in this dissertation.

1.2 Haskino

Figure 1.1 illustrates the system I have developed to advance the use of functional languages on the Arduino, known as Haskino (Grebe, 2017a) (Grebe, 2017b). The figure shows the capabilities that have been developed in each of three steps of research.

In step 1, I developed a shallow EDSL, as well as a deep EDSL, both written in Haskell (Grebe & Gill, 2016). They allow the end user to write a program on the host, while the computations specified by the program are executed on the Arduino using a firmware interpreter. The shallow EDSL allows the user to interactively program the Arduino, with intermediate results being returned to the host computer connected by USB cable. This interactive setup makes debugging and prototyping of new code and hardware much easier. The deep EDSL provides a way of outsourcing entire groups of commands and control-flow idioms to the Arduino. This allows a user's Haskell program to store a bytecode program on the board, then step back and let it run. Both of these EDSL methods use the remote monad design pattern to provide the key capabilities.

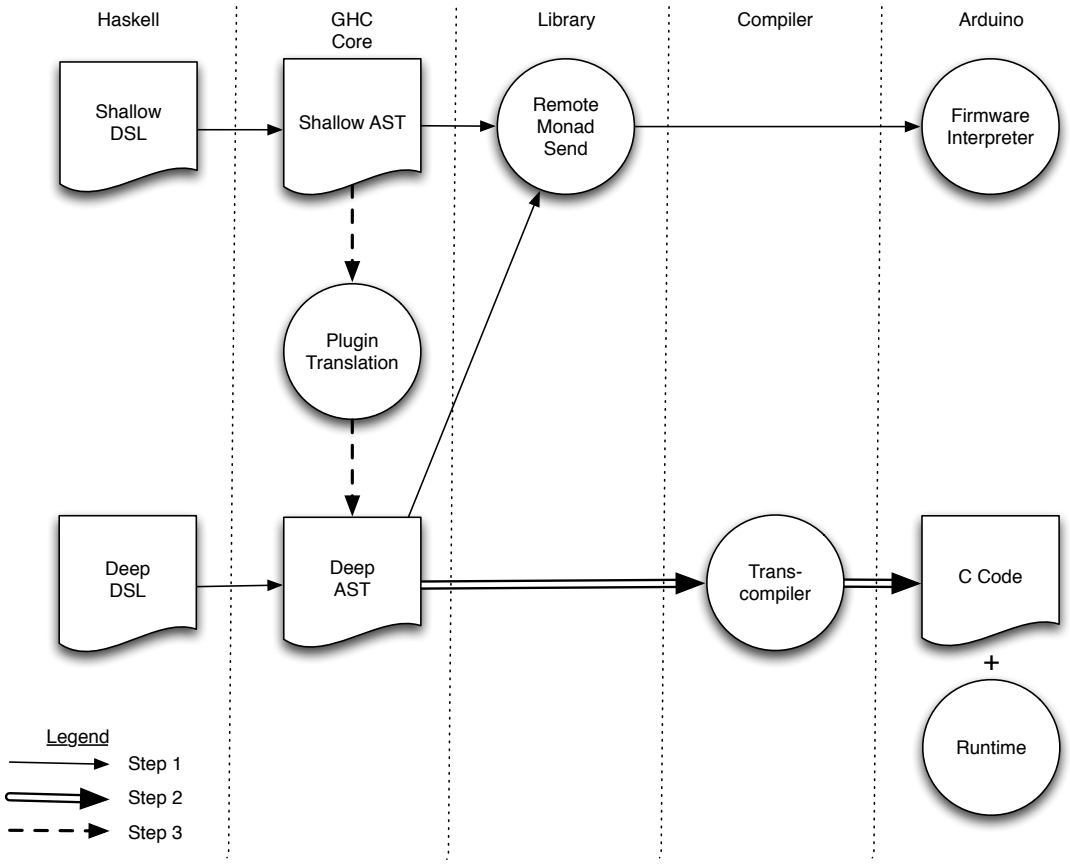

Figure 1.1: Haskino Overview

For step 2 of the research, Haskino was modified to use the same monadic code that is executed with the interpreter, but instead compile it into C code (Grebe & Gill, 2017). That C code may then be compiled and linked with a small runtime, to allow standalone operation of an executable with a smaller size than the interpreted code. This smaller size allows more complex programs to be developed and executed within the limited resources of the Arduino. In addition, the second stage of Haskino research added the concept of multi-threaded operation to the system. Programming embedded microcontrollers often requires the scheduling of independent threads of execution, specifying the interaction and sequencing of actions in the multiple threads. I added thread scheduling and inter-thread communication concepts to the EDSLs. In addition, I added schedul-

ing and communication capabilities enabling those concepts to both the firmware interpreter and the small run time.

In step 3 of the research, I explored enabling the user to write programs once in the shallow EDSL, and then have the code automatically translated to the deep EDSL by the Haskino system. The syntax of the shallow EDSL is much closer to standard Haskell than the syntax of the deep EDSL, using Haskell operators and control flow mechanisms. In addition, I wanted the user to be able to use the recursive style of iteration that is normally used in functional programming, as opposed to imperative programming's traditional `for` and `while` structures. To enable this style, the program translation must be able to translate tail recursive functions written in the shallow DSL into the `iterateE` structure used in the deep EDSL. To enable these translations, I have developed a plugin for the GHC compiler which is able to perform the translations using principled methods. With the translation system, the user is able to write the code once and use it with both the interpreted, interactive system, as well as the compiled, efficient system.

While many small to medium sized example programs have been developed during the course of performing the research, to better validate the results of the research, a larger, more complicated example was developed. This example is a version of the Haskino interpreter written in the shallow EDSL, which is then transformed to the deep EDSL by the compiler plugin, and then compiled to C and machine code with the Haskino system. This allowed the Haskino interpreter to be "bootstrapped" with Haskino itself.

Step 4 of the research examines a method to optimize the C code generated by the Haskino system. Most EDSLs which represent the target code as data structures inline the function calls in the language. For small embedded systems, this can be problematic, once again limiting the size of the programs that may be developed on the system. Haskino could be extended to generate non-inlined C code, once again through the GHC compiler plugin. Functions in the shallow EDSL may be marked by the programmer to enable or disable inlining. The compiler plugin translator will handle the translation of the non-inlined functions into deep DSL expressions which may then be compiled to separate C functions by the compiler.

1.3 Contributions

This dissertation makes the following contributions:

- It explores the design and implementation of Haskino, an embedded domain specific language (EDSL) for programming Arduino microcontrollers with Haskell. Haskino consists of a shallow EDSL with an interpreter, and a deep EDSL which is compiled, providng a gentler way of programming Arduino systems than the method used by the tools provided with the Arduino. Haskino also supports programming using multi-tasking and synchronization concepts normally required in small embedded systems. This dissertation presents details and rationale of many of the design choices that were made during the implementation.

- It demonstrates a method for performing translation of a shallow EDSL in Haskell to a deep EDSL (assuming the EDSL is based upon the Remote Monad library), using a set of principled transformation rules. This allows the user to write in a shallow EDSL which is automatically translated to the deep EDSL, providing syntax much closer to native Haskell.

- It demonstrates a method for performing translation of tail recursive function calls to an iterative structure in a Haskell EDSL (assuming a Remote Monad based EDSL), also using a set of principled transformation rules. This further enhances the capability to write in a native Haskell syntax, using recursive calls familiar to functional programmers.

- It investigates the design and implementation of a GHC compiler plugin that mechanizes the shallow to deep and recursive EDSL transformations. This automatic translation using the GHC compiler allows for an easy transition between interpreted debugging and compiled deployment by simply flipping a command line flag. The plugin is designed to be reusable with many monadic EDSLs through configuration based on the EDSL primitives, operators, and types. With the plugin, first the programer can prototype an idea using the interpreter, but with near the full power of Haskell, then use the automated translation mechanism to produce compiled code which is efficient in resources and performance.

- It examines resource utilization issues related to the compilation to C code from the Deep EDSL. The case study in Chapter 10 details the resource utilization issues with the code generated from transformed Haskino EDSL programs. To address these issues, a method for making significant reductions in resource usage through sharing is demonstrated.

1.4 Organization

The remaining portions of this dissertation are organized as follows.

Background

- Chapter 2 provides background material such that this dissertation is able to standalone. It provides an introduction to Arduino programming using traditional imperative methods, an introduction to the Remote Monad design pattern, details about the GHC compiler and it's plugin architecture, and introduction to the Worker-Wrapper transformation.

Haskino Design and Implementation

- Chapter 3 covers the design and implementation of the initial version of the Haskino system, implemented as a shallow EDSL with the Remote Monad design pattern.

- Chapter 4 deals with the conversion of Haskino to use a deep embedding in addition to a shallow embedding, enabling the addition of remote binding of computations.

- Chapter 5 describes the addition of scheduling to the Haskino DSLs and firmware, enabling the use of mutli-threaded programming commonly used in embedded systems.

- Chapter 6 details the trans-compiler developed for the Haskino system, which is able to transform the deep EDSL into C code to allow for efficient standalone operation of Haskino programs.

Transformations: Theory and Implementation

- Chapter 7 describes the theory behind the first of the two major transformations in Haskino, transforming a shallow EDSL into a deep EDSL.
- Chapter 8 covers the theory of the second of the Haskino transformations, transforming tail recursive code in the deep DSL into iterative structures.
- Chapter 9 details the implementation of the theory described in Chapters 7 and 8 as a GHC compiler plugin.

Case Studies

- Chapter 10 describes case studies built with Haskino, the main one being an implementation of a version of the Haskino firmware interpreter in Haskino itself. The chapter compares the performance and utilization statistics this bootstrapped interpreter with an implementation in native imperative Arduino C.
- Chapter 11 presents a third transformation, designed to deal with an issue highlighted in the interpreter case study, by eliminating duplicate generated code.

Closing

- Chapter 12 provides context of my research, detailing other systems which have dealt with functional programming on the Arduino, or with the mixture of shallow and deep EDSLs.
- Chapter 13 concludes the dissertation, and reflects on the development, as well as details future work to be done in this area.

Chapter 2

Technical Background

This chapter summarizes the supporting technologies which are relevant to the design and implementation of Haskino, so that this dissertation is able to be self-contained. First, background material on traditional software development for Arduino boards is discussed. Then, the concept of a Remote Monad is introduced, followed by a discussion of GHC, it's itermediate language Core, and it's plugin system which is used to implement Haskino's transformations. Finally, Worker-Wrapper transformations are discussed, which form the basis for the Haskino plugin transformations.

2.1 Arduino Background

Arduino microcontrollers vary in size from the Uno, which has a 16MHz clock rate, 2 KB of RAM, 32 KB of Flash, and 1 KB of EEPROM, to the Due which has a 84MHz clock rate, 96 KB of RAM, and 512 KB of Flash. The processors used on the Arduino boards are also not all of the same architecture, and include those from the AVR family on lower end boards such as the Uno, as well as lower end ARM processors on the higher end Arduino boards such as the Due. What these boards have in common, however, is a single API for programming them than spans the line of boards, and provides users a standardized interface for accessing the hardware. The Arduino series of microcontrollers are traditionally programmed using imperative C language programming with this standardized API. This API provide a set of programming primitives which are used to control the various hardware interfaces present in the Arduino microcontrollers.

Programming an Arduino board is all about controlling external hardware and responding to inputs from external hardware through pins of the microcontroller that are connected to internal peripherals. The Arduino boards provide easy to use connectors to these pins, allowing quick connection of the external hardware. The API for the Arduino is split into groups that handle the various types of pins on the microcontroller. The pins numbers that are in each group vary by the Arduino board type, however, there is some commonality between the boards. For example most boards have a build in LED on the board which is connected to pin 13.

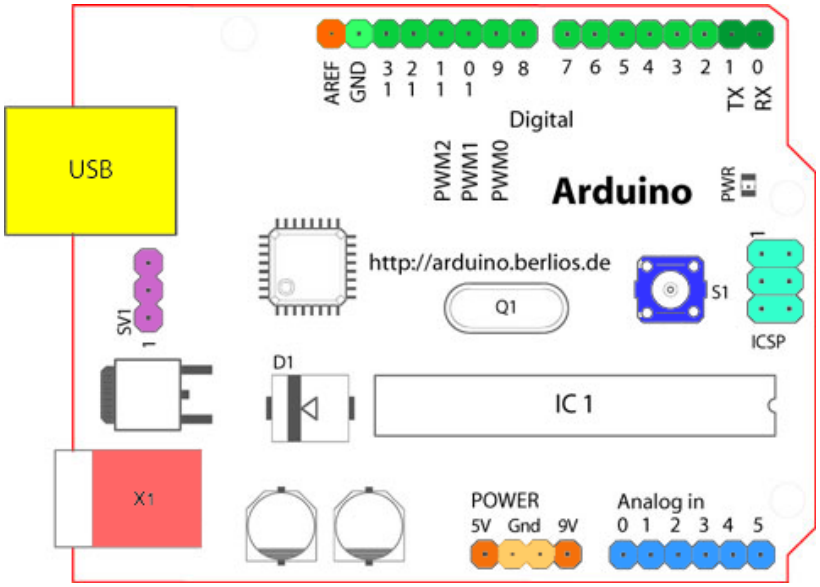

Figure 2.1: Arduino Uno Board

(arduino.cc, 2017)

Figure 2.1 shows the layout of an Arduino Uno board, the entry level board in the Arduino family, taken from the arduino.cc website (where it is licensed under a Creative Commons Attribution-ShareAlike 3.0 License). Pins used for digital I/O, analog output, and special purpose functions are shown in green on the top of the board. On the bottom of the board in blue, are pins used for analog input, as well as pins providing power (5V and ground). The USB connection is used to provide a serial port emulation, which is used to transfer programs to the board's flash memory, as well as for serial communication with a running program on the Arduino. The red connector at the

lower left of the board is used to provide external power, although the board may be powered via the USB connection as well.

2.1.1 Arduino General Purpose IO

The first basic type of pin on Arduino boards is a digital general purpose input/output (GPIO) pin. These pins are able to be set to either an input or an output, and this mode is controlled by calling the following API function:

```
void pinMode(uint8_t pin, uint8_t mode)
```

The `pin` parameter, which is used in many of the API functions, specifies the number of the pin to set the input/output mode of. The `mode` parameter of the function is set to `INPUT` or `OUTPUT` to specify if the pin is an input or output. There is also a third option, `INPUT_PULLUP`, which adds a pull-up resistor in addition to making a pin an input.

To read the value of a digital input pin, the user calls the following function:

```
int digitalRead(uint8_t pin);
```

It returns a value of `HIGH` (1) if the input pin has a digital high voltage (5V or 3.3V depending on the board), and a value of `LOW` (0) if the input pin has a digital low voltage (0V). Similar to the programming the digital input, to output a value on a digital output pin, the user can call the following function:

```
void digitalWrite(uint8_t pin, uint8_t val);
```

Like with the digital input function, the `val` parameter specifies if the output voltage driven on the output pin is `HIGH` (5V or 3.3V) or `LOW` (0V).

2.1.2 Arduino Time

The Arduino API also provides functions dealing with time. The user may read the amount of time since the board has been powered on, in either microseconds or milliseconds using:

```
unsigned long micros();
unsigned long millis();
```

In addition to reading the current time, the user may delay program execution for a specified duration, once again either in microseconds or milliseconds:

```
void delayMicroseconds(unsigned int us);
void delay(unsigned long);
```

2.1.3 Some Arduino Examples

Arduino C programs, also known as sketches, consist of two main functions. The first of these, setup(), is run by the Arduino kernel once to allow the program to initialize the hardware and software state. Then, the second function, loop(), is called repeatedly in an infinite loop.

The following example combines the digital output functions and the time functions into a program which blinks a LED once a second:

```
int ledPin = 13;   // LED connected to digital pin 13

void setup() {
    pinMode(ledPin, OUTPUT);   // sets the digital pin 13 as output
    }

void loop() {
    digitalWrite(ledPin, HIGH); // sets the LED to on
    delay(500);                 // delay for 0.5 seconds
    digitalWrite(ledPin, LOW);  // sets the LED to off
    delay(500);                 // delay for 0.5 seconds
    }
```

This second example demonstrates using a push button as a digital input to decide which of two LEDs to light.

```
int button =  7;     // Button connected to digital pin 7
int ledPin1 = 11;    // LED1 connected to digital pin 11
int ledPin2 = 12;    // LED2 connected to digital pin 12

void setup() {
    pinMode(button, INPUT);   // sets the digital pin 7 as input
    pinMode(ledPin1, OUTPUT); // sets the digital pin 11 as output
    pinMode(ledPin2, OUTPUT); // sets the digital pin 12 as output
    }

void loop() {
    int buttonVal;                          // variable for button state
    buttonVal = digitalRead(ledPin, HIGH); // reads the button state
    digitalWrite(ledPin1, buttonVal);      // set LED1 to button state
    digitalWrite(ledPin2, !buttonVal);     // set LED2 to not button state
    delay(100);                             // delay for 0.1 seconds
    }
```

2.1.4 Arduino Analog I/O

The Arduino boards are also capable of dealing with analog inputs and outputs as well. Analog input signals are converted by an analog-to-digital (A/D) converter in the Arduino processor to a 10 bit digital value, or a value from 0-1023. To read the value of a analog input pin, the user calls the following function:

```
int analogRead(uint8_t pin);
```

Certain of the pins on the Arduino are also able to output an analog value, not thru a digital-to-analog converter, but thru a technique known as Pulse Width Modulation, or PWM. With pulse width modulation, the hardware produces a square wave on the pin at a relatively high frequency, and varies the pulse width, or the on time, of the square wave. Changing the pulse width changes the average value of the signal, and is used to simulate a constant analog voltage. This may be used, for example, to vary the brightness of a LED connected to the analog output pin. To output a value on a analog output pin, the user can call the following function:

```
void analogWrite(uint8_t pin, int val);
```

The following example demonstrates the use of both analog input and analog output, allowing the user to control the brightness of an LED using a potentiometer.

```
int ledPin = 9;      // LED connected to digital pin 13
int aInPin = 3;      // potentiometer connected to analog pin 0

void setup() {
    pinMode(ledPin, OUTPUT);     // sets the pin as output
}

void loop() {
    int a;                       // place to store the read value
    a = analogRead(aInPin);      // read the analog input
    analogWrite(ledPin, a / 4);  // write the analog output, scaled 0-255
}
```

2.1.5 Higher Level Arduino Interfaces

Besides the low level interfaces of digital and analog inputs and outputs, the Arduino microprocessors also contain controllers which allow the control of higher level interfaces to transfer commands and data between the Adruino and external hardware. For example, the Arduino boards are capable of communicating on several serial data interfaces, including RS-232 serial, I2C, and SPI. The I2C and SPI interfaces are busses, and allow the Arduino controller to talk to multiple devices connected to the serial bus. The following example demonstrates writing to an EEPROM connected to a I2C bus.

```
int eeAddr = 44;      // Address of EEPROM on I2C bus

void eepromWrite(int addr, int count, uint8_t *data) {
    int i;                           // Loop counter for write
    Wire.beginTransmission(eeAddr); // transmit to device at I2C eeAddr
    Wire.write((addr >> 8) & 0x7F); // Write high byte of addr to EEPROM
    Wire.write( addr       & 0xFF); // Write low byte of addr to EEPROM
    for (i=0; i<count; i++) {
        Wire.write(data[i]);         // Write data byte to EEPROM
    }
}
```

The `Wire.beginTransmission` function is used to start transmitting data to a specific device on the I2C bus, and the `Wire.write` function is used to acutally write the data to the I2C bus. In this example, first two bytes are written which specify the memory address of where to write the data in the EEPROM, then the actual data is written.

In addition to accessing built in serial interface controllers, the Arduino API also provides library routines which combine basic digital and analog I/O with timers and interrupts to control such devices a stepper and servo motors. These are useful for the control of cyber-physical systems, such a small robotics and process control systems. The basic digital and analog I/O may also be combined to control more sophisticated devices that require multiple inputs and outputs. A common Arduino library example of such a device is a multi-line LCD display based on the Hitachi44780 controller. Many of these displays are available with an I2C control interface, but there are also variants that are controlled by 6-7 digital I/O pins as well. We will show examples of programming such devices later in this dissertation, in Section 5.4.2, Section 8.4, and Section 8.6.

2.1.6 Building and Running Arduino Programs

Arduino software may be developed with the provided Arduino IDE, or a developer may instead choose to use the component tools that are used behind the scenes by the IDE. Arduino C code is compiled and linked with a variant of the GCC compiler toolchain. The C code is compiled using `gcc`, then linked with the Arduino libraries using the GNU linker, `ld`. The object code output of the linker is then converted to a Intel hex format file. This hex format file is then downloaded to the Arduino board over the USB serial connection using a tool called AVRDUDE, or the AVR Download UploaDEr. Running on the Arduino board at the other end of the USB connection is a bootloader, which listens for program updates over the serial connection, then downloads the update and programs it to flash memory. Once the programming to flash is complete, the bootloader starts the execution of the newly downloaded program from flash.

Development on the Arduino boards, using the standard C development toolchain, is a repeated series of compiling, linking, downloading and flashing to the hardware, and then debugging. This can cause long development times, especially with new or unfamiliar hardware attached to the Arduino, so a method of quickly prototyping code would be a welcome addition.

2.2 Remote Monad

A **remote monad**(Gill et al., 2015) 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.

$$\texttt{send} \;::\; \forall\, a \;.\; \mathit{Remote}\; a \rightarrow \mathit{Local}\; a$$

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` type is the remote monad, and there are three remote primitives given here as an example. 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 remote monad may use a *weak* packet bundling that sends both commands and procedures one at a time, to be remotely executed. The design pattern in this case is a way of structuring remote procedure calls, but has no performance advantage. Alternatively, a remote monad may use

a *strong* packet bundling, which bundles together chains of commands, terminated by an optional procedure, which has the interesting result that is returned by the primitive. This improves performance, by reducing the communication overhead imposed by sending each packet individually. Further research has shown that there is a third alternative (Dawson et al., 2017). The *applicative* packet bundling uses the realization that in the use of `<*>`, the arguments on either side are independent of each other. This realization allows the bundling of multiple primitives, without restrictions, into a single packet, a structure for serialization that is also used by Haxl (Marlow et al., 2014). In addition to defining this third, more performant, bundling strategy, the further work has led to a generalized framework (Gill & Dawson, 2016) that encapsulates the design pattern.

We have built a number of libraries using the remote monad design pattern and framework. Blank Canvas (Gill & Scott, 2015) is our Haskell library that provides the complete HTML5 Canvas API, using a 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. Finally, Haskino is an application of the remote monad concept to programming embedded systems, with the packet bundling allowing remote execution with a bytecode interpreter.

2.3 Haskell and GHC

Haskell is a general purpose pure functional programming language, and it is widely used for work in Domain Specific Languages(Elliott & Hudak, 1997; Elliott et al., 2003; Axelsson et al., 2010, 2011; Svenningsson & Axelsson, 2013; Elliott et al., 2015; Hickey et al., 2014; Bracker & Gill, 2014). The principle Haskell compiler used in both research and industry is the Glasgow Haskell Compiler (GHC)(GHC Team, 2016). It lends itself nicely to my transformation work, providing a plug-in architecture which allows transformations and optimizations of programs.

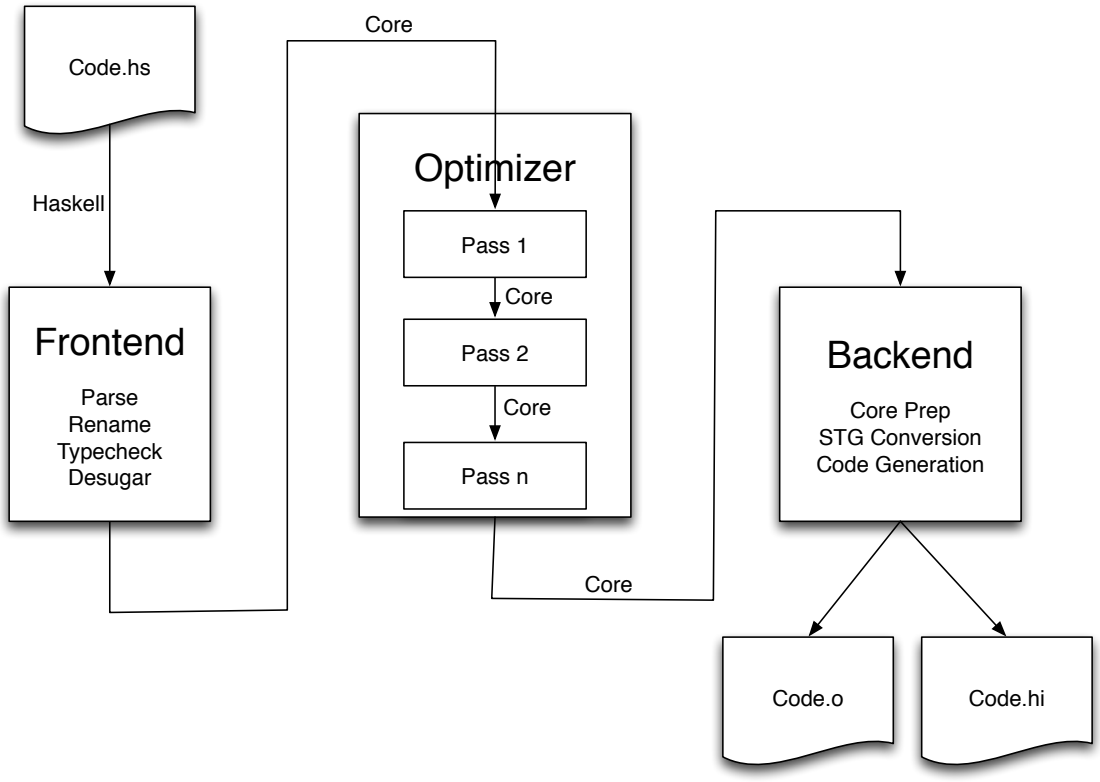

Figure 2.2: GHC Architecture

Figure 2.2 shows the architecture of GHC. GHC consists of a frontend, an optimizer, and a backend. This dissertation focuses on the optimizer, which is used for our transformations. The frontend parses the Haskell source, typechecks it, and desugars the Haskell into the first of GHC's three intermediate languages, Core (Peyton Jones & Santos, 1998). The optimizer consists of a series of passes which take a Core program as input, and produces an new, optimized version of Core as output. The compiler's backend takes the optimized Core, and coverts it to GHC's two other intermediate languages, first STG and then C--, in preparation for code generation. The backend outputs the generated object code, as well as an interface file for use in compiling other modules dependent upon the one under compilation.

2.3.1 GHC Core

GHC Core is an implementation of System F_C(Sulzmann et al., 2007), which is itself a extension of System F(Girard et al., 1989), adding support for non-syntactic type equality. The data types and constructors that make up GHC Core are shown in Figure 2.3. Core is a relatively compact language, compared to the large Haskell source language. Being small, and based on the mathematics of System F, it has been able to remain largely unchanged from it's original definition, even with large additions to the Haskell source language.

```
data ModGuts = ModGuts  _ :: [Bind], ...

data Bind b
  = NonRec b (Expr b)
  | Rec [(b, (Expr b))]

data Expr b
  = Var          Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Coercion Coercion

type Arg b = Expr b

type Alt b = (AltCon, [b], Expr b)

data AltCon
  = DataAlt DataCon
  | LitAlt  Literal
  | DEFAULT
```

Figure 2.3: Definition of the Core Intermediate Language

A Core program is made up of a list of bindings, either recursive (`Rec`), or non-recursive (`NonRec`). The bindings bind an identifier (`Id`) to an expression (`Expr`). A identifier may be

either a value or a type, and are unique. Identifiers may be used as part of an expression by using the `Var` constructor. Literals are specified with the `Lit` constructor.

Lambda application and abstraction are handled by the `App` and `Lam` constructors. Arguments to an application may be either terms or types, allowing for parametric polymorphism. The `Type` constructor is used when a type appears in the argument position in an application.

The `Let` constructor is used for both recursive and non-recursive let bindings. Term level let-bindings may be either recursive or non-recursive, while type let-bindings may only be non-recursive.

The `Case` constructor is used for expression pattern matching in the Core language, using the `Alt` and `AltCon` types to express the pattern alternatives. `Case` is also used to handle conditionals in the Core language, and for conditionals is just pattern matching over the boolean `True` and `False` constructors.

The `Cast` constructor handles wrapping expressions to change their types. The `Coercion` is the equality witness of the cast, and is generated by the type checker and manipulated by the the optimization passes.

Finally, `Tick`'s are used for expression annotation, usually for debugging and profiling. They are created by Haskell source level annotations.

2.3.2 Core Dictionaries

Haskell supports ad hoc polymorphism through it's type class feature (Wadler & Blott, 1989). This feature is supported in GHC Core through the use of dictionaries. A dictionary can be thought of as a tuple of length n, containing the n functions of a type class for a specific type instance.

Take as an example the function below, `add`, which adds two integers.

```
add :: Int -> Int -> Int
add x y = x + y
```

The addition function `(+)` is a function in the `Num` type class.

```
class  Num a  where
    (+) :: a -> a -> a
```

The Core for the add function has the form shown below. The Core here is shown in the format that is normally dumped by the compiler for debug, not in terms of the constructors from the last section.

```
add :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
add =
  \ (x :: GHC.Types.Int) (y :: GHC.Types.Int) ->
    GHC.Num.+ @ GHC.Types.Int GHC.Num.$fNumInt x y
```

The Core GHC.Num.+ function takes 4 arguments, two of which are the original integer arguments to (+). The other two arguments are the type argument, GHC.Types.Int, and the dictionary argument, GHC.Num.$fNumInt. This dictionary argument means, select the (+) field from the GHC.Types.Int dictionary for the GHC.Num type class.

2.3.3 GHC Compiler Plugins

GHC's compiler plugin architecture (GHC Team, 2016) allows the compiler user to modify or add to the passes that are performed in the optimizer. A plugin has the following type:

```
type Plugin = [CommandLineOption] -> [Pass] -> CoreM [Pass]
```

The plugin is passed any command line options that were specified at the compiler invocation, as well as the current set of Pass's currently scheduled to be performed by the optimizer. The plugin may then add, delete, or modify the order of passes that are performed by the optimizer. Each pass is defined as the following type:

```
type Pass = ModGuts -> CoreM ModGuts
```

An optimizer pass transforms a list of Core Bind's, which are part of the ModGuts data type given to the pass as input, into another list of Core Bind's in the returned ModGus. The ModGuts data structure is carried throughout all phases of the compiler, including plugin passes, and contains not only the Core of the module under compilation, but also a global reader environment of all in-scope symbols, GHC transformation rules, information about other modules imported to the one under compilation, and other information useful to the compiler pass. This plugin pass architecture is used to implement the DSL transformations described in Chapters 7 through 9.

2.3.4 GHC Rules

GHC also provides another mechanism for transforming programs known as rewrite rules(Jones et al., 2001). These rewrite rules allow the programmer to state properties that they know are true, and the compiler can then make use of those rules for optimization.

An example of one of these rewrite rules is shown below. GHC rewrite rules are stated using a RULES pragma. Following that is a name of the rule, in this case "map/map". The next line contains forall definitions, which indicate those variables which are universally quantified in the rule. Following that is the body of the rule, which gives the left hand side of the rule which the compiler must match, followed by an =, and the right hand side of the rule which should replace the left hand side if matched.

```
{-# RULES
    "map/map"
    forall f g xs.
              map f (map g xs) = map (f . g) xs
#-}
```

I have made use of these rewrite rules in the exploration of the shallow to deep DSL transformations. One of the restrictions of these rewrite rules is that the left hand side of the rule must be a function application. This limits the usefulness of using the rewrite rules in some of my transformations, however, I still make use of their syntax as a convenient transformation specification mechanism.

2.4 The Worker/Wrapper Transformation

The worker/wrapper transformation (Gill & Hutton, 2009; Jones & Launchbury, 1991) is a transformation that converts a computation of one type into a computation of another type (worker) wrapped by a function that converts between the types of the two computations (wrapper). These type of transformations are correctness preserving, and have been used in compilers and other applications for many years.

Assuming a function *f*, which has a the following form:

$$f = body$$

The right hand side, *body*, may have recursive calls to *f*. We can then replace the body with the wraper function, *wrap* applied to the worker function, *work*. The worker function itself is an application of the un-wrapper function, *unwrap* to the body of the original function.

$$f = wrap\ work$$

$$work = unwrap\ body$$

Where the body function is of type B, and the work function is of type A, the types of the worker/wrapper transformation are illustrated in Figure 2.4.

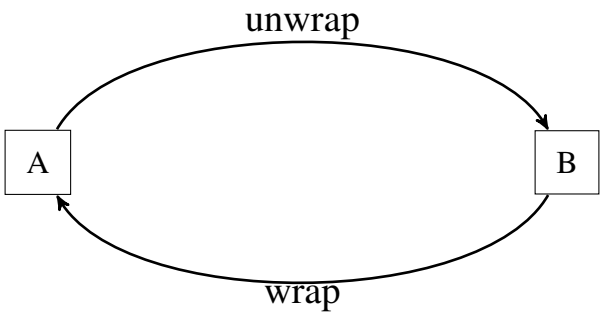

Figure 2.4: Worker-Wrapper Transformation

Applying this principle to our desired translations of shallow to deep EDSLs, and using the terminology of data representation (Hoare, 1972), we define the `rep` function which is the un-wrapper which moves from our normal Haskell types to the `Expr` representation, and `abs` which converts from the representation back to the abstract type. This is illustrated in Figure 2.5.

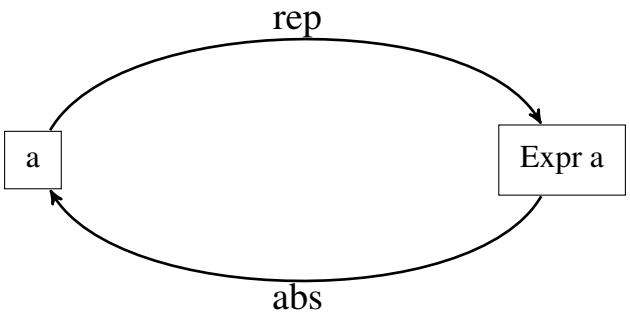

Figure 2.5: Expression Transformation

These conversions between the *abstract* type `a` and the *concrete* type `Expr a` depend on the worker-wrapper assumption that:

$$wrap \circ unwrap = id_A$$

Stating this in our DSL terms:

$$abs \circ rep = id_A$$

In the context of our expression language, this means that if we take a literal value in a base Haskell type such as Word8, move it the expression language with the `rep` function, then evaluate the resulting expression with the `abs` function, we will get the original value back.

It should be noted, than in my DSL work, I also use a worker-wrapper transformation in a monadic form, as shown in Figure2.6.

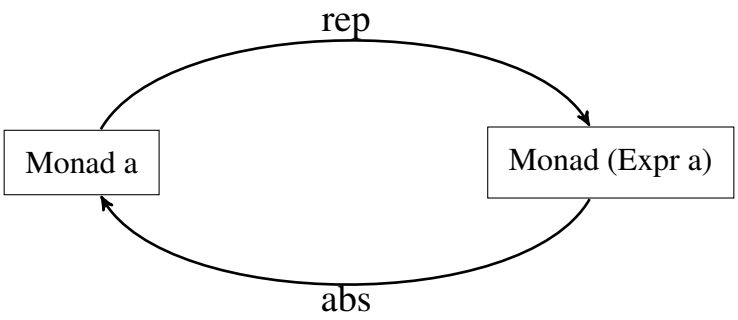

Figure 2.6: Monadic Transformation

Chapter 3

Remote Monads and Interpreters

The Arduino series of microcontrollers has previously been the target of of functional programming with Haskell. The hArduino package, written by Levent Erkök(Erkok, 2014), 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 was the starting point for Haskino.

3.1 The Arduino Remote Monad

The 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 strong remote monad `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 it 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 (Section 2.2), 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 shallow version of Haskino, this has the following signature:

```
digitalWrite :: Word8 -> Bool -> Arduino ()
```

The function takes the pin to write to and the boolean value to write, and returns a monadic value 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
```

Originally, the monad used in Haskino was defined using the original version of the remote monad library (Gill et al., 2015), parameterized over the ArduinoCommand and ArduinoProcedure data types.

```
newtype Arduino a = Arduino (RemoteMonad ArduinoCommand ArduinoProcedure a)
  deriving (Functor, Applicative, Monad)
```

The data types for ArduinoCommand and ArduinoProcedure were defined as GADTs as shown below, with only a subset of their actual constructors shown as examples.

```
data ArduinoCommand =
          ⋮
       DigitalWrite  Word8 Bool
     | AnalogWrite Word8 Word16
          ⋮
data ArduinoProcedure :: * -> * where
     DigitalRead    :: Word8 -> ArduinoProcedure Bool
     DelayMillis    :: Word32 -> ArduinoProcedure ()
          ⋮
```

The remote monad library was subsequently updated to not make the distinction between command and procedures using a different data type. Instead, it uses a separate function over a single data type for primitives to determine if the primitive in question has a known result, or in other words, was a command in the old nomenclature. The new monad currently used in Haskino is defined using the new remote monad library (Dawson et al., 2017), parameterized over the ArduinoPrimitive data type.

```
newtype Arduino a = Arduino (RemoteMonad ArduinoPrimitive a)
  deriving (Functor, Applicative, Monad)
```

The updated data type for the ArduinoPrimitive is defined as a GADT as shown below, with only a subset of the actual constructors shown as examples.

```
data ArduinoPrimitive :: * -> * where
          ⋮
     DigitalWrite   :: PinE -> Bool -> ArduinoPrimitive ()
     AnalogWrite    :: PinE -> Word16 -> ArduinoPrimitive ()
          ⋮
     DigitalRead    :: Word8 -> ArduinoPrimitive Bool
     DelayMillis    :: Word32 -> ArduinoPrimitive ()
          ⋮
```

The function knownResult in the remote monad package's KnownResult typeclass returns a Maybe type, with a `Just a` result for primitives with a known result (commands), and a Nothing for those without (procedures).

```
instance KnownResult ArduinoPrimitive where
          ⋮
  knownResult (DigitalWrite         ) = Just ()
  knownResult (AnalogWrite          ) = Just ()
          ⋮
  knownResult _                       = Nothing
```

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 = Arduino $ primitive $ DigitalWrite p b
```

To demonstrate the use of the shallow Haskino DSL, we return to the simple example presented in Section 2.1.3, this time written in the shallow version of the Haskino language.

```
example :: Arduino ()
example = withArduino False "/dev/cu.usbmodem1421" $ do
          let button1 = 2
          let button2 = 3
          let led = 13
          setPinMode button1 INPUT
          setPinMode button2 INPUT
          setPinMode led OUTPUT
          loop $ do
               a <- digitalRead button1
               b <- digitalRead button2
               digitalWrite led (a || b)
               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` functions are procedures, so they will be sent individually by the `send` function. The `digitalWrite` command following the two `digitalRead`'s will be bundled with the `delayMillis` procedure, and sent together by the `send` function.

3.2 Bytecode Interpreter and Protocol

I wanted to move from sending bundles of commands to the Arduino, to sending entire control-flow idioms, even whole programs, as large bundles. This can be done by using deep embedding technology, embedding both a small expressing language, and deeper Arduino primitives.

First however, 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 there was no requirement to maintain MIDI compatibility, I determined that it would be easier to develop a 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 the procedure computation.

Instead of 7 bit encoding, the frames are encoded with a 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 3.1 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,

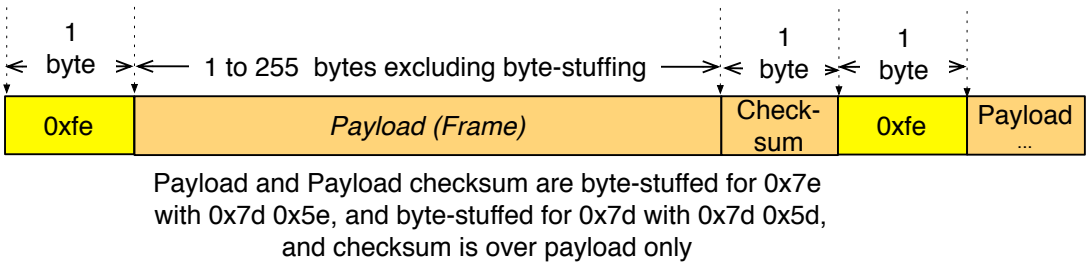

Figure 3.1: Haskino Framing

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, a maximum frame size for the protocol 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 Create-Task 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 at which to start running a task. Multiple tasks may be defined, and they run until 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. This command framing is show in Figure 3.2.

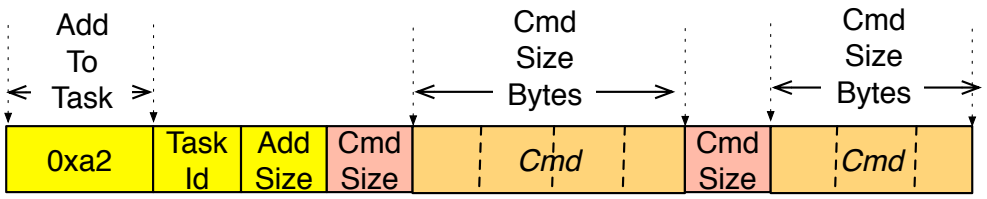

Figure 3.2: AddToTask Framing

This primitive scheduling capability is sufficient for the shallow version, which does not have the ability to chain results together remotely, so in reality these shallow "tasks" only make sense if they are composed of commands. It does, however, lay the framework for more sophisticated scheduling and concurrency mechanisms which will be presented in Chapter 5, after the deep embedding and the ability to chain remote computations are presented in Chapter 4.

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

Chapter 4

Remote Binding of Computations

To move towards my 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, I needed to move on from the Haskino shallow DSL used in Chapter 3. A deep embedding of the Haskino language allows a Haskino program to deal not just with literal values, but with complex expressions, and to define bindings that are used to retain results of computations remotely.

4.1 The Deep Extensions

To create a deeply embedded version of Haskino, the command and procedure monadic primitives are extended to take expressions as parameters, as opposed to simple values. This requires a new expression data type, `Expr`. For example, the `digitalWrite` command described earlier now becomes the `digitalWriteE` command:

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

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

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

4.1.1 The Expr Type

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. The `Expr` data type

is defined as a GADT, and a subset of it is shown below to demonstrate the type of constructors present in it.

```
data Expr a where
  LitUnit      :: Expr ()
  LitB         :: Bool -> Expr Bool
  LitW8        :: Word8 -> Expr Word8
    ⋮
  ShowW16      :: Expr Word16 -> Expr [Word8]
  ShowW32      :: Expr Word32 -> Expr [Word8]
    ⋮
  RefB         :: Int -> Expr Bool
  RefW8        :: Int -> Expr Word8
    ⋮
  NegW8        :: Expr Word8 -> Expr Word8
  AddW8        :: Expr Word8 -> Expr Word8 -> Expr Word8
    ⋮
  AndW8        :: Expr Word8 -> Expr Word8 -> Expr Word8
  OrW8         :: Expr Word8 -> Expr Word8 -> Expr Word8
    ⋮
  EqW8         :: Expr Word8 -> Expr Word8 -> Expr Bool
  IfW8         :: Expr Bool  -> Expr Word8 -> Expr Word8 -> Expr Word8
    ⋮
```

`Expr` is defined over types which are instances of several different type classes which provide a convenient API for users of Haskino to program with. The types that are currently part of the Haskino Expr language include boolean, signed and unsigned integers of length 8, 16 and 32, floats, as well as unit. Handling reads and writes from I^2C 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`, and it is also an instance of type classes used with the Haskino `Expr` type. To support using `Word8` lists in expressions, Haskino's expression language contains 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 to support debugging operations, and displaying data on Arduino attached displays.

The first type class that types use with the Haskino `Expr` type is the standard Haskell `Num` class, which provides standard arithmetic operations. The instance of `Num` for the `Expr Word8` type is show below. It translates the typeclass functions into constructors in the `Expr` data type.

```
instance Num (Expr Word8) where
  (+) x y = AddW8 x y
  (-) x y = SubW8 x y
  (*) x y = MultW8 x y
  negate x = NegW8 x
  abs x  = x
  signum x = SignW8 x
  fromInteger x = LitW8 $ fromInteger x
```

Several of the other type classes used with `Expr` are part of the Data.Boolean package (Elliott, 2013), which also provide a standardized interface. The first of these provides boolean operations for the `Expr Bool` type. The operations look similar to normal boolean operations, however they do not have the exact same names. For example, the name of the boolean `and` function is `(&&*)`, as opposed to the standard Haskell `(&&)` function.

```
instance Data.Boolean.Boolean (Expr Bool) where
  true  = LitB True
  false = LitB False
  notB  = NotB
  (&&*) = AndB
  (||*) = OrB
```

The second Data.Boolean class, Data.Boolean.Number.IntegralB, provides operations analogous to the standard Haskell Integral type. Once again, the instance for `Expr Word8` is shown below.

```
instance Data.Boolean.Numbers.IntegralB (Expr Word8) where
  div = DivW8
  rem = RemW8
  quot = QuotW8
  mod = ModW8
  toIntegerB e = ToIntW8 e
```

Comparison and conditionals of Haskino `Expr` types are provided by three Data.Boolean classes, `EqB`, `OrdB`, and `IfB`. As with the boolean operators, these operators are similar to standard Haskell operators, but have an appended asterisk, such as `>*` for greater than and `==*` for equals.

```
instance Data.Boolean.EqB (Expr Bool) where
  (==*) = EqB
instance Data.Boolean.OrdB (Expr Bool) where
  (<*) = LessB
instance Data.Boolean.IfB (Expr Bool) where
  ifB = IfB
```

As Haskino deals with hardware register manipulations, it is also convenient to define bit-wise operations over integers for Haskino `Expr` types. To this end, Haskino defines a new `Data.Boolean.Bits.BitsB` type class to handle those operations, similar to the standard Haskell `Data.Boolean.Bits` class. The instance for `Expr Word8` is shown below for this class.

```
instance Data.Boolean.Bits.BitsB (Expr Word8) where
  type IntOf (Expr Word8) = Expr Int
  (.&.) = AndW8
  (.|.) = OrW8
  xor = XorW8
  complement = CompW8
  shiftL = ShfLW8
  shiftR = ShfRW8
  isSigned = (_ -> lit False)
  bitSize = (_ -> lit 8)
  bit = ı -> 1 `shiftL` i
  setBit = SetBW8
  clearBit = ClrBW8
  testBit = TestBW8
```

Finally, Haskino defines a typeclass for Haskino specific operations, the main Haskino `Expr` typeclass, `ExprB`. The `lit` operator is used to lift standard Haskell values into the `Expr` type. The `remBind` function is used for remote binds which will be described in the next section, and `showE` is the show function mentioned earlier in this section, which translates Haskino `Expr` types in displayable lists.

```
class ExprB a where
    lit      :: a -> Expr a
    remBind  :: Int -> Expr a
    showE    :: Expr a -> Expr [Word8]

instance ExprB Word8 where
    lit = LitW8
    remBind = RemBindW8
    showE = ShowW8
```

The ExprB typeclass and Data.Boolean typeclass implementations were modified somewhat when the Shallow to Deep plugin transformations were developed. That design tradeoff is explored in Section 9.8,

4.1.2 Deep Allocations

The second component of the deep embedding is the ability to define remote references and bindings which allow us to use the results of one remote computation in another. For the first type of remote allocations, remote references, 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 (Expr ())
    modifyRemoteRef :: RemoteRef a -> (Expr a -> Expr a) ->
                       Arduino (Expr ())
```

The second type of remote allocations present in Haskino's deep embedding are remote binds. These terms in the `Expr` language are used to encode the monadic binds present in the source language. The `send` function is responsible for tracking these binds, and encoding them into the bytecode language which implements them.

4.1.3 Deep Conditionals

The final component required for the deep embedding is adding conditionals to the language. Haskino defines two types of conditional monadic structures, an If-Then-Else structure, and a iteration structure. The `IfThenElse` structure takes a `Expr Bool` expression, and returns a monadic expression from either the `Then` or the `Else` branch, with an `Arduino (Expr a)` type. The `iterateE` structure emulates various types of loops, and it takes an initial value for the loop variable, and a body function which returns an Either type composed of the loop variable type and the return type.

If the body returns a `(ExprLeft a)` value of the ExprEither type, then the iteration will continue, and the body will be called again. However, if the body returns a `(ExprRight b)` value of the ExprEither type, then the iteration will terminate, and `b` will be returned.

```
ifThenElse :: Expr Bool -> Arduino (Expr a) -> Arduino (Expr a) ->
              Arduino (Expr a)

data ExprEither a b where
    ExprLeft   :: (ExprB a, ExprB b) => Expr a -> ExprEither a b
    ExprRight  :: (ExprB a, ExprB b) => Expr b -> ExprEither a b

iterateE :: Expr a ->
            (Expr a -> Arduino (ExprEither a b)) ->
            Arduino (Expr b)
```

4.1.4 Shallow in Terms of Deep

Shallow remote monad commands may be defined in terms of their deep counterparts, allowing both to coexist in the deep embedded version. For example, after defining a simple `Expr ()` evaluation function, we can write the shallow version of digitalWrite in terms of the deep version:

```
evalExprUnit :: Expr () -> ()
evalExprUnit _ = ()

digitalWrite :: Word8 -> Bool -> Arduino (Expr ())
digitalWrite p b = evalExprUnit <$> (digitalWriteE (lit p) (lit b))
```

4.2 DSL Iteration Design Choices

As the Haskino DSL's were designed, the ability to chose what the form of the iteration structure would look like in the Deep EDSL presented itself. The original version of Haskino (Grebe & Gill, 2016) had a `while` iteration structure of the form:

```
while :: ExprB a =>
         Expr a -> (Expr a -> Expr Bool) ->
         (Expr a -> Expr a) -> Arduino () ->
         Arduino ()
```

This structure took an initial value, a function which tested the loop expression, a function which modified the loop expression each iteration, and a loop body function. This type of iterative structure worked for simple non-effectful iteration, which does not require a value to be returned.

However, Haskino is a monadic DSL, that deals with input and output from low level hardware, so it requires the ability to handle effects that occur in the body of an iterative structure. The next structure that was implemented in Haskino for iteration had the following structure.

```
whileE :: Expr a -> (Expr a -> Expr Bool) ->
          (Expr a -> Arduino (Expr a)) ->
          Arduino (Expr a)
```

In this version, the body function is monadic, allowing effects to be taken into account. However, it requires the loop variable and the return type to be of the same type. This posed difficulty in handling many of embedded examples I attempted such as the analogKey example from Chapter 8, and was not a general solution. It was however simpler to implement, since it did not require adding the concept of an Either type to the expression language.

These factors drove the design to use the form of iteration I have used in the examples in Chapter 8, which takes an initial value, and a body function which returns an Either type composed the loop variable type and the return type (Grebe et al., 2017). It has the following form.

```
iterateE :: Expr a ->
            (Expr a -> Arduino (ExprEither a b)) ->
            Arduino (Expr b)
```

This generalized structure allows each iteration of the loop to return either a new value for a loop variable, or a return value which may be of a different type.

Also, it should be noted that the earlier `whileE` structure can be implemented in terms of this new iterateE. It is also possible to implement a `loopE` structure which provides a deep analog of the `loop` structure used in the shallow version, and a replacement for the forInE which was used in earlier Haskino versions to provide iteration over lists, both written in terms of the iterateE structure as shown below.

```
whileE :: ArduinoIterate a a => Expr a -> (Expr a -> Expr Bool) ->
                     (Expr a -> Arduino (Expr a)) -> Arduino (Expr a)
whileE i tf bf = iterateE i ibf
  where
    ibf i' = do
        ifThenElseEither (tf i') (do
                                     res <- bf i'
                                     return $ ExprLeft res)
                                  (return $ ExprRight i')

loopE :: Arduino (Expr ()) -> Arduino (Expr ())
loopE bf = iterateE LitUnit (_ -> bf >> (return $ ExprLeft LitUnit))

forInE :: Expr [Word8] -> (Expr Word8 -> Arduino (Expr ())) ->
          Arduino (Expr ())
forInE ws bf = iterateE 0 ibf
  where
    ibf i = do
        _ <- bf (ws !!* i)
        ifThenElseEither (i `lessE` (len ws))
          (return $ ExprLeft (i+1))
          (return $ ExprRight LitUnit)
```

This allowed older code to be used with the new version, and also provides convenient shortcuts for common loop types.

4.3 The Unit Dichotomy

When originally moving from the shallow to the deep DSL, commands were left returning a standard unit type in the Arduino monad, `Arduino ()`, as they did in the shallow version of the command, as shown below:

```
digitalWrite  :: Word8 -> Bool -> Arduino ()

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

Also, in the original deep implementation, the IfThenElse construct did not return a value, but also returned `Arduino ()`. When an ifThenElse construct was added which was able to return values, two types of IfThenElse structures were present in the system, to remain consistent with the types that were returned from deep commands:

```
ifThenElseUnit :: Expr Bool -> Arduino () -> Arduino () ->
                  Arduino ()

ifThenElse :: Expr Bool -> Arduino (Expr a) -> Arduino (Expr a) ->
              Arduino (Expr a)
```

While this implementation proved functional, and did not require `()` to be a type in the expression language, it was not ideal. The handling of `()` as a special case caused the code used for shallow to deep transformations in the plugin (Chapter 9), to become overly complicated, requiring additional code to handle the `()` type differently than all others.

The solution to this issue was to make `()` a full fledged instance of the `ExprB` type class, and eliminate the special case handling. This allows the shallow to deep transformation of commands and procedures to be unified, as well as allowing the two types of IfThenElse structures to also be unified. This did require adding an `EXPR_UNIT` type to the bytecode langauage for the interpreter as well, but this required much less code and effort than the special handling for `()` in the GHC plugin.

4.4 Deep Protocol and Firmware

Changes to the Haskino protocol and firmware were also required to implement expressions, conditionals and remote allocation. Expressions are transmitted over the wire using a bytecode representation. Each operation is encoded as a two byte opcode with two fields. The first byte indicate the type of expression (currently Bool, Word8, Word16, Word32, Int8, Int16, Int32, [Word8], Float or Unit) and the second byte indicates the operation (literal, remote reference, addition, etc.). The ExprEither type is encoded in the type byte as well, with the most significant bit of the type byte indicating if it is ExprLeft, or ExprRight. 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 4.1 shows an example of encoding the addi-

tion of Word8 literal value of 4 with the first remote reference defined on the board, as well as a diagram of that expression being used in an analogWrite command.

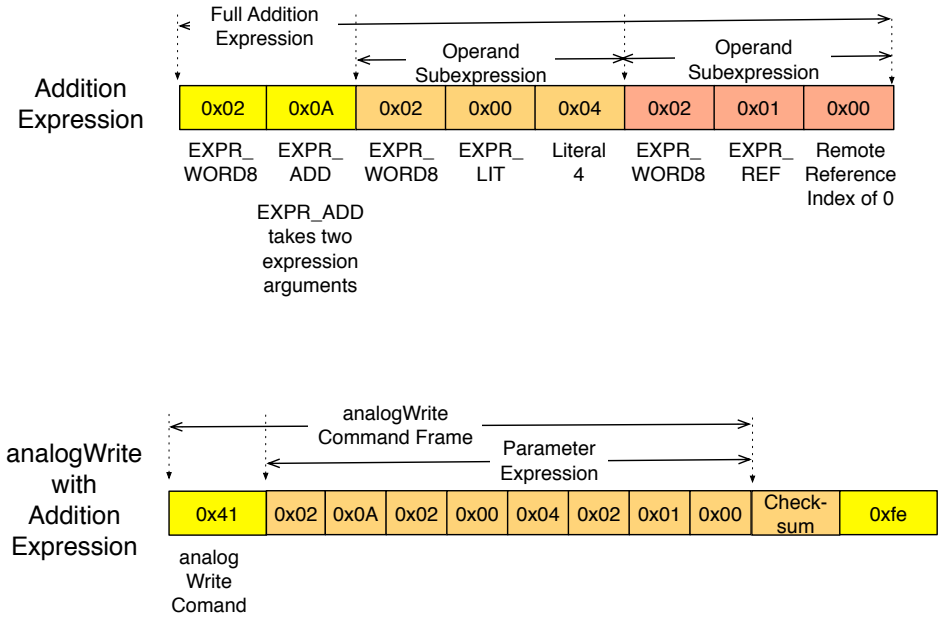

Figure 4.1: Example of Expression Encoding

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 iterateE structure. In 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 block(s) 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.

The Iterate command also returns an Either type, so the encoding of the command includes bytes which indicate the type of both the left and right components of the Either. (Iteration and it's use of Either types is explored in more detail in Chapter 8). To support our recursive transformations, the IfThenElse type also has a variant that returns an Either type. Therefore, the encoding of IfThenElse also includes bytes indicating the type of the left and right components. If the

IfThenElse being encoded does not return an either, the left and right type bytes are simply set to the same value. Figure 4.2 shows the encoding of both conditional types.

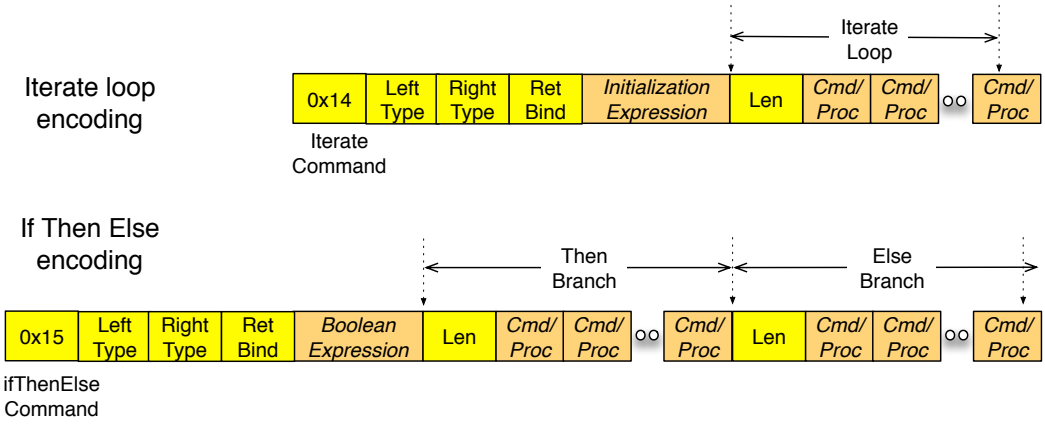

Figure 4.2: Protocol Packing of Conditionals

4.5 Deep Example

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

```
exampleE :: Arduino (Expr ())
exampleE = withArduino True "/dev/cu.usbmodem1421" $ do
           let button1 = 2
           let button2 = 3
           let led = 13
           setPinModeE button1 INPUT
           setPinModeE button2 INPUT
           setPinModeE led OUTPUT
           loopE $ do
                a <- digitalReadE button1
                b <- digitalReadE button2
                digitalWriteE led (a ||* b)
                delayMillis 100
```

This deep example looks very similar to the shallow example in structure. The bindings `a` and `b`, which were previously stored on the host, are now kept on the Arduino. The Haskino firmware 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.

4.6 Debugging

The original shallow version of the Haskino DSL provided rudimentary debugging capabilities through a `debug` primitive.

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

The `debug` primitive is a special case within the monadic `send` function, using the host Haskell `show` functions and `liftIO` to display values to the console. 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 deep version of the Haskino DSL instead includes a `debugE` primitive whose expression parameters are evaluated on the Arduino:

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

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 Arduino.

One of the key features of Haskino is that the same monadic code may be used for both interpreted and compiled (Chapter 6) versions. This allows for quick prototyping with the tethered, interpreted version, and then allows for the compiling of 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.

When the shallow to deep translation was added to Haskino through a compiler plugin (Chapter 7), it was desirable to change the debug primitives to use similar types so that they could be automatically converted without requiring special cases. This was required as the Haskino language does not have `Char` or `String` types, but instead uses a `[Word8]`. Therefore, the shallow debug primitive was changed to use the following type instead.

```
debug :: [Word8] -> Arduino ()
```

As the shallow debug primitive now takes a `[Word8]` parameter instead of a `String`, it does require a modified show function to be used in the shallow version to convert Haskino types into `Word8` lists. The types of the new show routines for the shallow and deep Haskino versions are shown below.

```
showB :: Show a => a -> [Word8]

showE :: ExprB a => Expr a -> Expr [Word8]
```

This extends the commonality of the debugging routines. If the shallow DSL code is written using the new `debug` and `showB`, it may be automatically translated by the plugin, and will work with all of the Haskino variants, the shallow interpreter, deep interpreter, and deep compiler with one version.

Chapter 5

Scheduler

Programming 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.

To address these requirements, Haskino has been developed to include the capability of multi-threaded operation. The shallow 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 had no way to store results from one procedure for use in another command or procedure, the usefulness of these tasks was severely limited. The shallow version of Haskino extended the ability of tasks, allocating remote references to store the results of a procedure and use that result in a subsequent command or procedure. The shallow version was, however, still limited to running a single task to completion.

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

As an example of multiple threads running in a Haskino program, the following program is presented.

```
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
    loopE $ do
        takeSemE semId
        writeRemoteRef i 0
        iterateE 0 (\ x ->
            ifThenElseEither (x <* 3)
                (do digitalWriteE led true
                    delayMillisE blinkDelay
                    digitalWriteE led false
                    delayMillisE blinkDelay
                    return $ ExprLeft $ x + 1)
                (return $ ExprRight $ lit () ))

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

initExample :: Arduino ()
initExample = do
    let led = 13
    createTaskE 1 $ myTask1 led
    createTaskE 2 myTask2
    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.

5.1 Scheduling the Interpreter

The Haskino firmware interpreter runs byte code on the Arduino, the byte code having been generated on the host and transmitted to Arduino using the remote monad function `send`. 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.

Haskino defines two conditional structures, an If-Then-Else structure and a Iterate structure. Each of these structures contains the concept of execution of basic code blocks within the interpreter. 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 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, Iterate, this is all that is required. The other control structure requires 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 Else branch.

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 `delayMillisE` 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. Another delay procedure, `delayMicrosE`, exists for those cases where the programmer wishes a task to have a short delay without the possibility of being interrupted, and executing this procedure will not cause a task reschedule. The second procedure which may interrupt execution of a task is described in the next section.

5.2 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 synchronization between tasks, Haskino provides several methods. First, the RemoteReference class provides atomic storage methods that can be used to pass data between Haskino tasks. RemoteReference's provide an API analogous to the Haskell IORef, allowing a remote reference to be read, written, or modified.

Haskino also provides binary semaphores for synchronization between Haskino tasks. A binary semaphore may be given by one task by executing the `giveSem` procedure, while the task that wants to synchronize with the first task can do so by executing the `takeSem` procedure. When a task executes 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. However, the task is not suspended in this case, 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
```

5.3 Firmware Scheduler Details

The interpreter tracks the scheduling and state of tasks in the system using several data structures. The first of these is the task control block as shown below:

```
typedef struct task_t
    {
    struct task_t      *next;
    struct task_t      *prev;
    struct context_t   *context;
    byte                id;
    uint16_t            size;
    uint16_t            currLen;
    uint16_t            currPos;
    uint32_t            millis;
    bool                ready;
    bool                rescheduled;
    byte                data[];
    } TASK;
```

The task control blocks are kept in a linked list, and have a pointer to a context structure which will be described next, and a 8 bit id which is used to reference the task in creation, deletion, and scheduling commands. When the task is created by the `createTask` primitive, it's maximum bytecode size is calculated by the `send` function on the Haskell host as it encodes the task EDSL

code into bytecode. The bytecode is stored in the `data` field of the task control block, and the `currLen` field is updated as AddToTask firmware protocol commands are received, since the task bytecode may be too large to fit in one protocol packet.

The remaining fields of the task control block are used by the scheduler to run and schedule the tasks. The `ready` boolean is used to indicate if the task is ready to run or not, and the `rescheduled` boolean indicates if the task was running but has been rescheduled. The `millis` field indicates for a rescheduled task what time it is next scheduled to run, which is used to implement the delay commands. The `currPos` is used during task execution to indicate the current position in the byte code being executed, what would commonly be called the program counter.

The second structure used for task management is the context structure. This is kept in a separate structure from the task structure due to the fact that there is a default context, which is used to execute primitives sent from the host individually that are not part of any defined task.

```
typedef struct context_t
    {
    TASK               *task;
    BLOCK_STATUS        blockStatus[MAX_BLOCK_LEVELS];
    int16_t             currBlockLevel;
    int16_t             recallBlockLevel;
    uint16_t            bindSize;
    byte               *bind;
    bool                left;
    } CONTEXT;
```

The `context` structure contains a pointer back to it's owning task, which in the case of the default context will be `NULL`. The `bindSize` field is set at context creation, and indicates the number of bind variables to be allocated for this context, and those allocated binds are stored in the buffer pointed to by the `bind` context element. The number of binds required for the task is calculated by the `send` function on the host as the tasks EDSL is translated into bytecode. Three fields of the context structure are used by the scheduler to track execution within code blocks when a tasks execution may be interrupted as described in Section 5.1. The final field of the context, `left`, is used for processing of `ExprEither` type values as part of the `iterateE` structure execution.

Finally, binary semaphores are tracked by the scheduler using another simple structure. The semaphore structure contains a boolean flag which indicates if the semaphore has been given (`full`), and a pointer to a task which is waiting on the semaphore.

```
typedef struct semphore_t
    {
    bool full;
    TASK *waiting;
    } SEMAPHORE;
```

5.4 Examples

To better illustrate the utility of the Haskino system with a multithreading program, we present two slightly more complex examples. The first example demonstrates using Haskino to program multiple tasks with asynchronous timing relationships. The second example demonstrates using tasks to simplify a program which would otherwise require hardware status busy waiting.

5.4.1 Multiple LED Example

In this first 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 loop turning the LED on, delaying the specified time, turning the LED off, and then again delaying the specified time.

```
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
```

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 is 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, the scheduler then call the Arduino runtime `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 `delayMillisE` call, when the scheduler will be called and will call `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.

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

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

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

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

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.

5.4.2 LCD Counter Example

In the second example, an Arduino board has an LCD display shield attached, which in addition to the display also has a set of six buttons (up, down, left, right, select, and enter). The buttons are all connected to one pin, and the analog value read from the pin determines which button is pressed. The example will display a signed integer counter value on the LCD display, starting with a counter value of zero. If the user presses the up button, the counter value will be incremented and displayed. Similarly, if the user presses the down button the counter value will be decremented and displayed.

The main function of the program, `lcdCounterTaskInit`, creates two Haskino tasks, one for reading the button, and another for updating the display. It also creates a remote reference which will be used for communicating the button press value between the tasks.

```
lcdCounterTaskInit :: Arduino ()
lcdCounterTaskInit =  do
      let button = 0
      setPinModeE button INPUT
      taskRef <- newRemoteRef $ lit (0::Word16)
      createTaskE 1 $ mainTask taskRef
      createTaskE 2 $ keyTask taskRef button
      -- Schedule the tasks to start immediately
      scheduleTaskE 1 0
      scheduleTaskE 2 0
```

The key task waits for a button press, reading the analog value from the button input pin until it is less than 760 (A value greater than 760 indicates that no button is pressed). The value read from the pin, which indicates which button was pressed, is stored in the remote reference used to communicate between tasks. At this point, the semaphore is given by the task. It then waits for the button to be released, and repeats the loop.

```
keyTask :: RemoteRef Word16 -> Expr Word8 -> Arduino ()
keyTask ref button = do
    let readButton :: RemoteRef Word16 -> Arduino ()
        readButton r = do
            val <- analogReadE button
            writeRemoteRef r val
    releaseRef <- newRemoteRef $ lit (0::Word16)
    loopE $ do
        writeRemoteRef ref 760
        -- wait for key press
        while ref (\ x -> x >=* 760) id $ do
            readButton ref
            delayMillisE 50
        giveSemE semId
        writeRemoteRef releaseRef 0
        -- wait for key release
        while releaseRef (\ x -> x <* 760) id $ do
            readButton releaseRef
            delayMillisE 50
```

The main task sets up the LCD (with the `lcdRegisterE` call), and creates a remote reference which will track the counter value. It thens turns on the LCD backlight, and writes the initial counter value to the display. At this point it then enters the main loop, waiting for the the key task to give the semaphore. When it receives the semaphore, it reads the key value from the remote

reference. Based on the value, it either increments the counter, decrements the counter, or does nothing. The counter value is then read from the remote reference and the display is updated with its value.

This second example has demonstrated using a remote reference in conjunction with a semaphore to communicate between Haskino tasks.

```
mainTask :: RemoteRef Word16 -> Arduino ()
mainTask ref = do
    lcd <- lcdRegisterE osepp
    let zero :: Expr Int32
        zero = 0
    cref <- newRemoteRef zero
    lcdBacklightOnE lcd
    lcdWriteE lcd $ showE zero
    loopE $ do
        takeSemE semId
        key <- readRemoteRef ref
        debugE $ showE key
        ifThenElse (key >=* 30 &&* key <* 150)
            (modifyRemoteRef cref (\ x -> x + 1)) (return ())
        ifThenElse (key >=* 150 &&* key <* 360)
            (modifyRemoteRef cref (\ x -> x - 1)) (return ())
        count <- readRemoteRef cref
        lcdClearE lcd
        lcdHomeE lcd
        lcdWriteE lcd $ showE count
```

5.5 Comparing Shallow to Deep

Table 5.1 summarizes the major differences that were found between the shallow and deep implementations. In the shallow 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 shallow 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 shallow version, due to the host interaction, is only limited by the larger host memory.

Table 5.1: Comparison of Shallow and Deep Embedding using Interpeter

	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

5.6 Cutting the Cord

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

Chapter 6

Compiler

The interpreted version of the Haskino DSL provides a quick turnaround Arduino development environment, including features for easy debugging. However, it has a 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, as described in Section 5.6. 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, a compiler was developed 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 :: FilePath -> Arduino () -> IO ()
```

6.1 Compiler Structure

The compiler processes the monadic code in a similar fashion to the way that the remote monad `send` function does for the interpreted version. Instead of reifying the GADT structures which represent the user programs into Haskino interpreter byte code, the compiler instead generates C code.

Each task in the program, including the initial task which consists of the code in the top level monadic structure, is compiled to C code using the `compileTask` function. The `compileTask` function makes use of the compiler's core function, `compileCodeBlock`, to recursively compile the program. The top level code block for the task, may contain sub-blocks for Iterate and IfThenElse control structures present in the top level block. A State monad is used by the compiler to track generated task code, tasks which are yet to be compiled as they are discovered in compilation of other task blocks, and statistics such as the number of binds per task, which are used for storage allocation as described in Section 6.4.

```
data CompileState = CompileState {
                      level :: Int
                    , intTask :: Bool
                    , ix :: Int
                    , ib :: Int
                    , cmds :: String
                    , binds :: String
                    , refs :: String
                    , forwards :: String
                    , cmdList :: [String]
                    , bindList :: [String]
                    , tasksToDo :: [(Arduino (Expr ()), String, Bool)]
                    , tasksDone :: [String]
                    , errors :: [String],
                    , iterBinds :: [(Int, Int)] }

compileTask :: Arduino (Expr ()) -> String -> Bool ->
               State CompileState ()

compileCodeBlock :: Bool -> String -> Arduino a -> State CompileState a
```

Expressions and control structures are compiled into their C equivalents, with calls to Haskino runtime functions for expression operators that are not present in the standard Arduino runtime library. ArduinoPrimitive's are likewise translated into calls to either the Arduino standard library, or to the Haskino runtime.

In the following sections, we will explain the C code which is generated by executing the `compileProgram` function on the initExample program which was shown as the opening example in Chapter 5.

6.2 Initialization Code Generation

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 iteration 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 code generated for Haskino initialization for this semaphore example follows:

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

void loop() {
    }

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

The `setup()` function serves three purposes. First, it initializes the memory management of the Haskino runtime, which is described in Section 6.7. 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 `setup()` 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 6.5, is started by calling the `startScheduler()` function.

6.3 Task Code Generation

The monadic code passed in each `createTaskE` call in the Haskell code is compiled into a C function, named `taskX()`, where X is the task ID number which is also passed to `createTaskE` (not the name of the monadic function). As an example, the code for the first task from the semaphore example is shown below:

```
void task1() {
    pinMode(13,1);
    while (1) {
        int bind0;
        takeSem(0);
        bind0 = 0;
        while(1) {
            if (bind0 < 3) {
                digitalWrite(13,1);
                delayMilliseconds(125);
                digitalWrite(13,0);
                delayMilliseconds(125);
                bind0 = bind0 + 1;
            } else {
                break;
            }
        }
    taskComplete();
    }
```

6.4 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 compilation of their newRemoteRef calls cause the following global allocations in the generated code (prior to any task functions) :

```
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. They are named similarly to remote references, with a name of the form bindX, where X is the id number of the bind assigned by the compiler. In the example, there is one Word8 bind in `myTask2`, used inside of the while loop:

```
t <- readRemoteRef loopCount
```

Its allocation as the local variable bind0 may be seen in the following code:

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

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

The `task2()` generated code above also demonstrates the initialization of the RemoteReference ref0 to its initial value, 0, in the first statement of the generated task. The remote reference ref0 is then incremented in each iteration, making use of the bind0 bind variable. This code also demonstrates the use of the debugging features of Haskino (discussed in Section 4.6), by outputting the loop count contained in bind0, with the `debug()` call.

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, which is the final type of storage allocated by the compiler, and it's structure is show in detail in Section 6.6. 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 three tasks from the semaphore example.

```
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];
```

The address of the allocated TCB, as well as the size of the allocated stack are passed to the task creation calls, as can be seen from the creation call for task1 shown below:

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

6.5 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 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.

6.6 Runtime Structure Detail

Analogous to the structures used by the firmware interpreter to track tasks and their scheduling, the compiler runtime's kernel uses task control blocks (TCBs) to manage the task scheduling.

```
typedef struct tcb_t
    {
    struct tcb_t       *next;
    byte                id;
    void               (*entry)(void);
    uint32_t            millis;
    bool                hasRan;
    bool                ready;
    uint16_t            stackPointer;
    uint16_t            stackSize;
    byte                stack[];
    } TCB;
```

Like the interpreter, the kernel keeps track of the TCBs as a linked list, and the `id` of the task used in the create and schedule commands is stored in the TCB. As was stated in Section 6.4, the local binds are allocated as stack variables. As we saw in Section 6.4, the stack is allocated as part of the TCB, and the TCB also contains fields which indicate the stack size, as well as storage space for the task stack pointer to be stored when a running task is interrupted by the scheduler. Also like the interpreter TCB, the `ready` field indicates if the task is ready to run, and if not, the `millis` field indicates at what millisecond tick it will be ready. Finally, the `hasRan` field is used to determine if the task has ran before. When the task is ran for the first time, the `entry` field will define what program location it will run from.

6.7 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 4.1.1). In both cases the garbage collection scheme is simple, with memory elements being freed when an asso-

ciated 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 program's 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 simply 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.

6.8 Comparing Interpreted and Compiled Size

It has been stated that the Haskino compiler makes more efficient use of the small amount of storage space available on the Arduino Uno boards than does the Haskino intepreter. Table 6.1 shows the amount of space used by both the Haskino interpreter, and the runtime used by the compiler, without any user program present. Both the raw size, and the percentage of the available resource are shown.

Table 6.2 shows the percentage of available Uno Flash and RAM used by the example programs from Section 5.4.1 (Example 1) and Section 5.4.2 (Example 2). The number of buffers available

	Haskino Interpreter	Haskino Runtime
Flash Size	31124 bytes	3052 bytes
RAM Size	901 bytes	437 bytes
Uno Flash Usage	95.0%	9.3%
Uno RAM Usage	44.0%	21.3%

Table 6.1: Interpreter and Runtime Storage Sizing with no user program

for dynamic memory management in the runtime is user configurable. The unoptimized numbers for the runtime reflect the default allocation, where the optimize reflect values customized for the specific program. These numbers are for hand written Deep EDSL code, not for those that are automatically translated, as will be presented in Chapter 7 and Chapter 9.

	Haskino Interpreter	Haskino Runtime
Example 1 Flash Usage	95.0%	14.5%
Example 1 Unoptimized RAM Usage	56.9%	70.8%
Example 1 Optimized RAM Usage	-	45.0%
Example 2 Flash Usage	95.0%	30.4%
Example 2 Unoptimized RAM Usage	151.6%	70.8%
Example 2 Optimized RAM Usage	-	47.9%

Table 6.2: Interpreter and Runtime Storage Sizing for Example Programs

Note that for Example 2, the LCD Counter example, the RAM requirements for the interpreted version of the program exceeds the memory available on a Uno board, due to the size of the generated byte code for the tasks. This program was tested using an Arduino Mega 2560 board which has 8 Kbytes of RAM, as opposed to the Uno's 2 Kbytes. However, the compiled version fits comfortably within the Uno's 32 Kbytes of flash, and 2 Kbytes of RAM.

While the size of the interpreter means that large programs may not be implemented with it in their entirety, it may still be used to prototype and debug smaller portions of more complicated programs. For example, it may be used to prototype code for interfacing to new hardware, where the hardware interface may not be well understood. Once the interface section is prototyped with the interpreter, the entire program may then be developed with the compiler.

Chapter 7

Shallow to Deep Translation

The shallow to deep transformation of a program written in a monadic EDSL that I developed uses a worker-wrapper based transformation to move from a shallowly to a deeply embedded language. To demonstrate my transformation, I return to our simple example with it's shallow EDSL syntax of:

```
let button1 = 2
let button2 = 3
let led = 13
loop $ do
    a <- digitalRead button1
    b <- digitalRead button2
    digitalWrite led (a || b)
    delayMillis 100
```

The simple example in the Deep ESDL syntax is:

```
let button1 = 2
let button2 = 3
let led = 13
loopE $ do
    a <- digitalReadE (lit button1)
    b <- digitalReadE (lit button2)
    digitalWriteE (lit led) (a ||* b)
    delayMillisE (lit 100)
```

The `lit` operations lift a basic Haskell type into the Expr expression type, and the `||*` operator is the logical or operation between two values of the `Expr Bool` type.

Writing even this simple example in the deeply embedded style presents challenges to the programmer, as opposed to the shallowly embedded style, which is more idiomatic Haskell. The overhead to writing in the deeply embedded style becomes even greater when using conditionals

and iteration. It would be preferable that the programmer be able to write in the shallowly embedded style, and let the compiler transform it to the deeply embedded style automatically. For now, I will show how this simple example may be written in the shallow version, and automatically converted to the deep version, and will deal with conditionals and iteration later in section Section 7.2 and Section 8.

7.1 Basic Transformation

In the following steps in the transformation, we will omit the initial let expressions, and concentrate on the loop body. First, we will de-sugar the do notation, and get the following form:

```
loop (
  digitalRead button1 >>=
    (\ a -> digitalRead button2 >>=
      (\ b -> digitalWrite led (a || b))) >>
        delayMillis 100)
```

In the first step of the transformation, we will convert each of the shallow commands and procedures into their deep versions, inserting worker-wrapper (Section 2.4) `abs` and `rep` function calls to maintain the types of the overall computation. The `rep` function moves a value from the basic Haskell type to one of the Expr type, and the abs function has the opposite effect, moving a value from a Expr type to a basic Haskell type. The `rep` is equivalent to the `lit` function of the `ExprB` typeclass, and may remain in the transformed code, while the `abs` function should never actually be evaluated in the transformed code.

```
rep :: ExprB a => a -> Expr a
rep w = lit w

abs :: Expr a -> a
abs _ = error "Internal error: abs called"
```

In the following descriptions of transformations, the transformations will be described in the style of GHC rewrite rules (Jones et al., 2001). In some cases the rules given would not be valid for GHC, as the left hand side is not a function application, however the syntax of the rules provides a convenient representation of the transformations.

The first transformation presented is of primitives, which return values. Each use of a shallow primitive `prim` will be transformed to use a deep version, `primE`. These shallow and deep versions have the forms:

```
prim :: a1 -> ... -> an -> Arduino b
primE :: Expr a1 -> ... -> Expr an ->
         Arduino (Expr b)
```

The term `prim` represents a generic shallow primitive, and `primE` represents a generic deep procedure. A specific transformation will be needed for each of the actual procedures in the DSL, but this generic procedure is used to show the form of the transformation. This transformation is achieved using the following rule.

```
forall (arg1 :: a1) ... (argn :: an).
prim arg1 .. argn
  =
abs <$> (primE (rep arg1) ... (rep argn))
```

Applying this transformation step to our example program, we get the following:

```
loopE (
  abs <$> digitalReadE (rep button1) >>=
    (\ a -> abs <$> digitalReadE (rep button2) >>=
      (\ b -> abs <$> digitalWriteE (rep led)
                           (rep (a || b)))) >>
       abs <$> delayMillisE (rep 1000))
```

Now that we have the worker-wrapper operators have been placed in the code, the next step in the transformation involves moving the rep operators inside of expressions, transforming the shallow functions over standard Haskell types into deep functions over types in the Expr data type. We refer to these transformations as “rep push” operations, pushing the rep operators to the interior of expressions. The instance of this type of transformation used in our simple example is for the boolean `or` operator, and the rule for the transformation is:

```
forall (b1 :: Bool) (b2 :: Bool).
rep (b1 || b2)
  =
(rep b1) ||* (rep b2)
```

After applying the `rep` push transformation rule to the example, the expression is transformed from a `Bool` type into a `Expr Bool` type as shown below.

```
loopE (
  abs <$> digitalReadE (rep button1) >>=
    (\ a -> abs <$> digitalReadE (rep button2) >>=
      (\ b -> abs <$> digitalWriteE (rep led)
                      ((rep a) ||* (rep b)))) >>
        abs <$> delayMillisE (rep 1000))
```

Now that we have moved the `rep` functions inside of expressions, we can apply the next rule of the transformation, starting to move the abs operators closer to the rep operators to achieve fusion. This rule is a variant of the third monad rule, and has the form:

```
forall (f:: Arduino (Expr a)) (k:: a -> Arduino b).
     (abs <$> f) >>= k
            =
     f >>= k . abs
```

Applying this monadic rule to the example, we move the `abs` operators through the two monadic binds, changing them to a composition of the continuation with the `abs`.

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a -> digitalReadE (rep button2) >>=
      (\ b -> abs <$> digitalWriteE (rep led)
                    ((rep a) ||* (rep b))) . abs
         ) . abs >>
        abs <$> delayMillisE (rep 1000))
```

Having moved the abs operators through the binds, with the next rule we would like to move the abs operators inside of the lambdas. The transformation to do this has the following form:

```
forall (f :: Arduino a).
(\ x -> f[x]) . abs
  =
(\ x' -> let x=abs x' in f[x])
```

The notation `f[x]` represents the usage of the binding x somewhere inside the function f. When this rule is applied to the example, two let expressions are inserted into the lambda expressions, as show below:

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a' -> let a = abs a' in
      digitalReadE (rep button2) >>=
        (\ b' -> let b = abs b' in
          abs <$> digitalWriteE (rep led)
                        ((rep a) ||* (rep b)))) >>
        abs <$> delayMillisE (rep 1000))
```

The let expressions may then be eliminated by replacing instances of `a` and `b` in the body of the lambdas with `(abs a')` and `(abs b')` respectively. This will result in:

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a' -> digitalReadE (rep button2) >>=
        (\ b' -> abs <$> digitalWriteE (rep led)
                        ((rep (abs a')) ||*
                         (rep (abs b'))))) >>
        abs <$> delayMillisE (rep 1000))
```

Now, with the `rep` and `abs` applications correctly positioned, one final simple transformation is required. The rep-abs combinations may be fused by the following rule.

```
forall x.
rep(abs(x))
  =
x
```

Applying the fusion rule, our transformed code has the following form:

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a' -> digitalReadE (rep button2) >>=
        (\ b' -> abs <$> digitalWriteE (rep led) ( a' ||* b'))) >>
        abs <$> delayMillisE (rep 1000))
```

There are two calls to `abs` left in the transformed code. As the primitives they are associated with are joined by the `>>` operator, not the `>>=` operator, and Haskell uses lazy evaluation, the return values of the functions that `abs` is applied to will never be evaluated. Therefore, they may be eliminated, and we have achieved of goal with the translation as shown below.

```
loopE (
  digitalReadE (rep button1) >>=
    (\ a' -> digitalReadE (rep button2) >>=
        (\ b' -> digitalWriteE (rep led) ( a' ||* b'))) >>
        delayMillisE (rep 1000))
```

The transformations described in the example in this section, and which are implemented in the plugin (Chapter 9), currently cover monadic code written with the higher level Haskell functions >>= and >>. They do not handle other higher level Haskell monadic functions such as `mapM`. The worker-wrapper transformation techniques used in this section could be extended to define rules for transforming instances of mapM and other higher order functions.

7.2 Transformation of Conditionals

Conditionals in deeply embedded DSLs normally take the form of functions over three arguments, one for the boolean test, and one each for the then and else branch of the conditional. Writing the conditionals in this form, as opposed to the normal Haskell if-then-else form, is another case where writing code for a deeply embedded DSL is inconvenient. The transformations presented here once again allow the program author to write in a shallowly embedded DSL form, like standard Haskell coding, and have the program automatically transformed to the deep EDSL form.

There are two types of conditionals which must be transformed. The first of these are conditionals where the then and else expressions are of the main data type of the EDSL. Once again, using our example of the Haskino language, these are terms of the `Arduino` monad type. In the Haskino language, this type of conditional function has the following type:

```
ifThenElseE :: ExprB a => Expr Bool ->
                          Arduino (Expr a) ->
                          Arduino (Expr a) ->
                          Arduino (Expr a)
```

Another small example code section which deals with three button inputs and two LED outputs will be used to demonstrate the conditional transformation:

```
a <- digitalRead button1
b <- if a
     then do
        digitalWrite led1 True
        digitalRead button2
     else do
        digitalWrite led2 True
        digitalRead button3
```

The main transformation of the conditional is similar to the command and procedure transformations we performed in Section 7.1. The rule syntax for the transformation is as follows:

```
forall (b :: Bool) (m1 :: ExprB a => Arduino a)
                   (m2 :: ExprB a => Arduino a).
if b then m1 else m2
  =
abs <$> ifThenElseE (rep b) (rep <$> m1)
                            (rep <$> m2)
```

Applying this rule to our example, after also applying the command and procedure transformations from Section 7.1, the shallow code containing the conditional is transformed into the following:

```
a <- digitalRead button1
b <- abs <$> ifThenElseE (rep a)
              (rep <$> do
                digitalWrite (rep led1) (rep True)
                digitalRead button2)
              (rep <$> do
                digitalWrite (rep led2) (rep True)
                digitalRead button3)
```

At this point, two more manipulation rules need to be added to the transformation toolbox. The first is a rule which will push the `fmap` application of `rep` through the monadic binds in the then and else branches. It has the following form:

```
forall (f :: Arduino a) (k :: a -> Arduino b).
    rep <$> (f >>=  k)
          =
    f >>= \ x -> rep <$> k x.
```

After the application of this rule, along with the other rules described in Section 7.1, the conditionals example is transformed to the following:

```
a' <- digitalReadE button1
b' <- ifThenElseE (a')
        (do
           digitalWriteE (rep led1) (rep True)
           rep <$> (abs <$> digitalReadE
                                (rep button2)))
        (do
           digitalWriteE (rep led2) (rep True)
           rep <$> (abs <$> digitalReadE
                                (rep button3)))
```

Now the final rule required is the `fmap` analog to the rep-abs fusion rule we used earlier in Section 7.1, which will fuse the `rep` and `abs` functions in the then and else branches.

```
forall (m :: Expr a => Arduino a).
rep <$> (abs <$> m)
  =
m
```

After applying this rule, the example is as follows:

```
a' <- digitalReadE button1
b' <- ifThenElseE (a')
        (do
           digitalWriteE (rep led1) (rep True)
           digitalReadE (rep button2)))
        (do
           digitalWriteE (rep led2) (rep True)
           digitalReadE (rep button3)))
```

The other form of conditional found in many deeply embedded DSLs is a conditional over the expression language. In the Haskino language, this conditional has the following type:

```
ifB :: ExprB a => Expr Bool ->
                  Expr a -> Expr a -> Expr a
```

Transformation of an if-then-else expression written in a shallowly embedded form to this deeply embedded conditional requires only one transformation rule, as shown below. Following this application rule, the rep-push rules described in Section 7.1 may be used to further reduce the expressions in the boolean test, as well as the then and else branches of the expression.

```
forall (b :: Bool) (t :: ExprB a => a)
                   (e :: ExprB a => a).
if b then t else e
  =
abs $ ifB (rep b) (rep t) (rep e)
```

Chapter 8

Iteration and Recursion Transformation

An Arduino C programmer would use `for` or `while` loops for programming iteration, however, a Haskell programmer would use recursion for the same task. The Haskino deep EDSL provides a iterateE structure for iteration, but as the goal of my research is to provide relatively idiomatic Haskell syntax to the programmer, using it is unsatisfying. Instead, it would be better to be able to translate tail recursive functions in the shallow EDSL into functions using the iterateE structure automatically, as has been done with conditionals and the other shallow components of the DSL.

8.1 First Recursion Example

Starting with a typical iteration example on the Arduino, we will blink a LED a specified number of times in Haskino.

```
led = 13
button1 = 2
button2 = 3

blink :: Word8 -> Arduino ()
blink 0 = return ()
blink t = do
    digitalWrite led True
    delayMillis 1000
    digitalWrite led False
    delayMillis 1000
    blink $ t-1
```

We would like to transform recursive functions of the type `Expr a -> Arduino(Expr b)` into functions which use an imperative iteration loop. We enable the transformation by creating a data type, `Iter`. This type indicates on a specific iteration of the loop, if the function should return

(it is "Done"), or if it should perform the computation associated with the next iteration of the loop (it needs to "Step").

```
data Iter a b
    = Step a
    | Done b
```

A Haskell function is also defined which performs the iterative loop. This function will not be used in the final implementation, but is defined to allow us to demonstrate the transformation method in the shallow version of the DSL. The iteration function, `iterLoop`, takes the initial value of the input argument, and a function which is able to perform a single step of the iteration that returns either a Step value of the input type, or a Done value of the output type.

```
iterLoop :: a -> (a -> Arduino (Iter a b)) ->
            Arduino b
iterLoop iv stepF = do
    result <- stepF iv
    case result of
        Step va -> iterloop va stepF
        Done vb -> return vb
```

The transformation is started by adding a wrapper function to insert the `iterLoop` function. The worker function, `blink` is formed by applying the Done constructor to the body of the original function, in which the pattern matching notation has been removed, replacing it with conditional notation.

```
blink :: Word8 -> Arduino ()
blink a = iterLoop blinkI a

blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI t = Done <$>
              if (t == 0)
              then return ()
              else do
                digitalWrite led True
                delayMillis 1000
                digitalWrite led False
                delayMillis 1000
                blink $ t-1
```

We now apply the first rule of our recursion transformation to the function, which is used to move the Done constructor through any conditionals:

```
forall f x.
    Done <$> if b then f else g
        =
    if b then (Done <$> f) else (Done <$> g)
```

Applying this rule to the example function gives:

```
blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI t = if (t == 0)
              then Done <$> return ()
              else Done <$> (do
                  digitalWrite led True
                  delayMillis 1000
                  digitalWrite led False
                  delayMillis 1000
                  blink (t-1)
```

We now introduce the other basic rule of the recursive transformation to move the Done constructor call to the end of the bind chain.

```
forall f g.
    Done <$> (f >>= g)
        =
    f >>= \ x -> Done <$> g x.
```

Applying this rule repeatedly we obtain:

```
blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI t = if (t == 0)
              then Done <$> return ()
              else do
                  digitalWrite led True
                  delayMillis 1000
                  digitalWrite led False
                  delayMillis 1000
                  Done <$> (blink (t-1))
```

Finally, in the branches where the recursive function call is present, we can apply the following rule to eliminate the recursive call, `f`, instead inserting the Step constructor.

```
forall x.
    Done <$> (f x)
        =
    Step <$> (return x)
```

With the example, and applying the rule, it then becomes:

```
blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI t = if (t == 0)
              then Done <$> return ()
              else do
                  digitalWrite led True
                  delayMillis 1000
                  digitalWrite led False
                  delayMillis 1000
                  Step <$> (return (t-1))
```

Finally, we use the following rule involving the `Done` constructor, and an equivalent one for the `Step` constructor.

```
forall x.
    Done <$> (return x)
        =
    return (Done x)
```

This moves the constructor inside of the return, and leaves us with the final transformed version:

```
blinkI :: Word8 -> Arduino (Iter Word8 ())
blinkI t = if (t == 0)
              then return (Done ())
              else do
                  digitalWrite led True
                  delayMillis 1000
                  digitalWrite led False
                  delayMillis 1000
                  return (Step (t-1))
```

8.2 Translating to Haskino Iteration

The example transformation of the last section was demonstrated on the shallow version of the DSL for clarity. We would now like to replace the `iterLoop` function which is written in Haskell, with the Haskino iteration primitive. This primitive has the following type:

```
iterateE :: Expr a ->
            (Expr a -> Arduino (ExprEither a b)) ->
            Arduino (Expr b)
```

The Haskino expression language also has an `ExprEither` type, as was discussed in Section 4.1.3, which will be used instead of the `Iter` type, and is defined as:

```
data ExprEither a b where
    ExprLeft   :: (ExprB a, ExprB b) =>
                  Expr a -> ExprEither a b
    ExprRight  :: (ExprB a, ExprB b) =>
                  Expr b -> ExprEither a b
```

As the iterateE and ExprEither are defined in the deep version of Haskino, it is preferable to do the recursive transforming after first transforming the example from the shallow language to the deep using the transformation from Section 7.1. Doing so gives us the following for the example from the previous section:

```
blinkE :: Expr Word8 -> Arduino (Expr ())
blinkE t =
        ifThenElseE (t ==* rep 0)
             (return (rep ()))
             (do
                digitalWriteE (rep led) (rep True)
                delayMillisE (rep 1000)
                digitalWriteE (rep led) (rep False)
                delayMillisE (rep 1000)
                blinkE  (t - (rep 1))
```

We now can use the method we demonstrated in the previous section, using `iterateE` instead of `iterLoop`, and the `ExprEither` type instead of the Iter type. We also need to replace the conditional used in the function, as `ifThenElseE` is only defined over one type, and instead we will use the `ifThenElseEither` which is defined over two types.

```
ifThenElseEither :: (ExprB a, ExprB b) =>
                     Expr Bool ->
                     Arduino (ExprEither a b) ->
                     Arduino (ExprEither a b) ->
                     Arduino (ExprEither a b)
```

Applying the recursion transformation method, using the `ExprLeft` constructor in place of the `Step` constructor, and `ExprRight` in place of `Done` we get:

```
blinkE :: Expr Word8 -> Arduino (Expr ())
blinkE t = iterateE t blinkEI

blinkEI :: Expr Word8 ->
                Arduino (ExprEither Word8 ())
blinkEI t =
        ifThenElseEither (t ==* rep 0)
             (return (ExprRight (rep ())))
             (do
               digitalWriteE (rep led) (rep True)
               delayMillisE (rep 1000)
               digitalWriteE (rep led) (rep False)
               delayMillisE (rep 1000)
               return (ExprLeft (t - (rep 1)))
```

8.3 Second Recursion Example

The second example deals with another form of iteration that is typical in systems that deal with hardware response. In this example we want a function that waits for a button to be pressed. This is detected by the return from a Haskino read of a digital pin becoming True. The function would be written in a shallow, tail recursive style as follows:

```
wait :: Arduino ()
wait = do
  b <- digitalRead button1
  if b then return () else wait
```

The method of transformation from Sections 8.1 and 8.2 works for functions with one argument, but not with functions with zero arguments as in this example. To transform functions of this type require us to first transform the function into one that takes a parameter of type `Expr ()` which is not used in the body of the function, but allows us to use the iterateE construct, and the ExprEither type, both of which are parameterized over two types. Adding the argument, and transforming the function from shallow to deep, we have:

```
waitE :: Ardruino (Expr ())
waitE = waitE' (lit ())

waitE' :: Expr () -> Arduino (Expr ())
waitE' _ = do
  b <- digitalReadE button1
  ifThenElseE b (return (lit ())) (wait (lit ()))
```

Applying the recursive transformation method from Section 8.1, we are able to transform this second small recursive example to the following:

```
waitE :: Ardruino (Expr ())
waitE = waitE' (lit ())

waitE' :: Expr () -> Arduino (Expr ())
waitE' t = iterateE t waitE'I

waitE'I :: Expr () -> Arduino (ExprEither () ())
waitE'I _ =
    b <- digitalReadE button1
    ifThenElseEither (b)
             (return (ExprRight (rep ())))
             (return (ExprLeft (rep ())))
```

The same method can be used to transform the recursive functions which are used as the top level loop in a typical Arduino program. Returning to our first simple example from Section 7.1, we may now write it recursively as:

```
progLoop :: Arduino ()
progLoop = do
    a <- digitalRead button1
    b <- digitalRead button2
    digitalWrite led (a || b)
    delayMillis 1000
    progLoop
```

Using the methods described in the previous section, this will be translated to:

```
progLoopE :: Arduino (Expr ())
progLoopE = progLoopE' (lit ())

progLoopE' :: Expr () -> Arduino (Expr ())
progLoopE' t = iterateE t progLoopE'I
```

```
progLoopE'I :: Arduino (Expr (Either () ()))
progLoopE'I _ = do
    a <- digitalReadE (lit button1)
    b <- digitalReadE (lit button2)
    digitalWriteE (lit led) (a ||* b)
    delayMillisE (lit 1000)
    return (ExprLeft (lit ()))
```

8.4 Third Recursion Example

As the third example, I present a transformation of recursion which demonstrates both a recursive function which returns a non-unit value, and the ability to greatly simplify a deep EDSL function by being able to write it in the shallow EDSL.

A common peripheral used on the Arduino is a small LCD display, and some of these display units also have a set of five buttons that may be used to indicate Up, Down, Left, Right, and Select. A press of one of these buttons is detected by the user program by reading a 16 bit analog input, where each single button press is denoted by a range of values. The shallow version of a recursive function which waits for one of the keys to be pressed, and returns an 8 bit unsigned integer corresponding to the button pressed is shown below:

```
analogKey :: Arduino Word8
analogKey = do
    v <- analogRead button2
    case v of
         | v < 30  -> return $ keyValue KeyRight
         | v < 150 -> return $ keyValue KeyUp
         | v < 350 -> return $ keyValue KeyDown
         | v < 535 -> return $ keyValue KeyLeft
         | v < 760 -> return $ keyValue KeySelect
      _            -> analogKey ()
```

The following is the resulting deep version after shallow to deep and recursive transformations:

```
analogKeyE :: Arduino (Expr Word8)
analogKeyE = analogKeyE' (lit ())

analogKeyE' :: Expr () -> Arduino (Expr Word8)
analogKeyE' t = iterateE t analogKeyE'I
```

```
analogKeyE'I :: Expr () ->
                Arduino (ExprEither () Word8)
analogKeyE'I _ = do
  v <- analogReadE button2
  ifThenElseEither (v <* 30)
    (return (ExprRight (lit (keyValue KeyRight))))
    (ifThenElseEither (v <* 150)
      (return (ExprRight (lit (keyValue KeyUp))))
      (ifThenElseEither (v <* 350)
        (return (ExprRight (lit (keyValue KeyDown))))
        (ifThenElseEither (v <* 535)
          (return (ExprRight (lit (keyValue KeyLeft))))
          (ifThenElseEither (v <* 760)
            (return (ExprRight (lit (keyValue KeySelect))))
            (return (ExprLeft (lit ()))))))))
```

With this example, we can see the advantage of using the pattern matching notation in the shallow version, as opposed to the more verbose deep notation, especially with the deeply nested conditionals present in the deep version.

8.5 Mutual Recursion

The recursion examples and methods that I have shown so far have all been of functions which call themselves. The recursion methods I have described may be extended to cover mutually tail recursive functions as well, and in this section I will describe those extensions. To walk through these recursive transformation methods, I will use a common basic example of mutual tail recursion, which is a recursive method of calculating if a number is even or odd. This is example is shown below written in shallow Haskino style.

```
isEven :: Word8 -> Arduino Bool
isEven 0 = return True
isEven n = isOdd $ n - 1

isOdd :: Word8 -> Arduino Bool
isOdd 0 = return False
isOdd n = isEven $ n - 1
```

To start the translation, we return to the `Iter` data type that we defined in Section 8.1. To enable the handling of mutually tail recursive functions, we extend the data type by adding a `Int`

parameter to the Step constructor. This additional parameter will allow us to specify which of the mutually tail recursive functions should be called to when we are making a recursive call (or "Step").

```
data Iter a b
    = Step Int a
    | Done b
```

In addition to adding the extra parameter to the `Iter` data type, we also add an `Int` parameter to the iteration function, `iterLoop`, and to the step function, `stepF`, that we defined in Section 8.1.

```
iterLoop :: Int -> a -> (Int -> a -> Arduino (Iter a b)) ->
            Arduino b
iterLoop i iv stepF = do
    result <- stepF i iv
    case result of
        Step i' va -> iterloop i' va stepF
        Done vb -> return vb
```

Now the original functions can both be defined in terms of a common function which executes the iteration function, passing in as our new `Int` parameter an index that indicates which of the original functions was called.

```
isEven :: Word8 -> Arduino Bool
isEven n = isEvenOdd 0 n

isOdd :: Word8 -> Arduino Bool
isOdd n = isEvenOdd 1 n

isEvenOdd :: Int -> Word8 -> Arduino Bool
isEvenOdd i n = iterLoop i n isEvenOddI
```

The step function created by the transformation is then made up of a sequence of conditionals which test the index passed to it, and compute the body of the appropriate recursive function based on that index. For our even/odd example, this is illustrated below:

```
isEvenOddI :: Int -> Word8 -> Arduino (Iter Word8 Bool)
isEvenOddI i n = do
    if i == 0
    then
        Body of recursive function 1 (isEven)
    else
        Body of recursive function 2 (isOdd)
```

Once this structure has been setup, we may proceed with applying the rules defined in Section 8.1 to insert, move, and transform the `Done` primitives to `Done` and `Step` primitives in each of the recursive function bodies. The only required modification to the rules is changing the step insertion rule to handle any of the recursive functions, inserting the corresponding index of the recursive function as a step parameter.

```
forall x.
    Done <$> f_index x
        =
    (Step index) <$> (return x)
```

Doing this in our even/odd example results in the final step function.

```
isEvenOddI :: Int -> Word8 -> Arduino (Iter Word8 Bool)
isEvenOddI i n = do
    if i == 0
    then
        if n == 0
        then return $ Done True
        else return $ Step 1 (n - 1)
    else
        if n == 1
        then return $ Done False
        else return $ Step 0 (n - 1)
```

8.6 Mutual Recursion State Machine

When constructing embedded systems software, a state machine is a common mechanism used. Mutual tail recursion provides a clean implementation of a state machine, with each mutually recursive function representing computation associated with a state, and the mutual tail recursive call a transition to the next state.

To demonstrate the usefulness of Haskino in programming such state machines, and as a further example of the mutual tail recursion transformation, I will present an example of a simple state machine using the displays and buttons that were discussed in Section 5.4.2 and Section 8.4. Also, where the first mutual recursion example used a shallow example, as I did for the first self recursive

example, this example will now move to using the deep embedding in the transformation, with the Haskino iterative function (`iterateE`) and type (`ExprEither`).

This example will use the six key keyboard on the LCD display/keyboard to cause state translations in the state machine. Upon entering a state, the state number and the key which was pressed to cause the state transition will be displayed on the LCD display. Upon power-up, the state machine will enter State1. From there, any key press will cause a transition to State2. To move between State2 and State3, the left and right keys are used. Pressing the select button in either State2 or State3 will return to State1. Any other key press in State2 or State3 will keep the machine in it's current state. The state machine diagram for this example is shown in Figure 8.1.

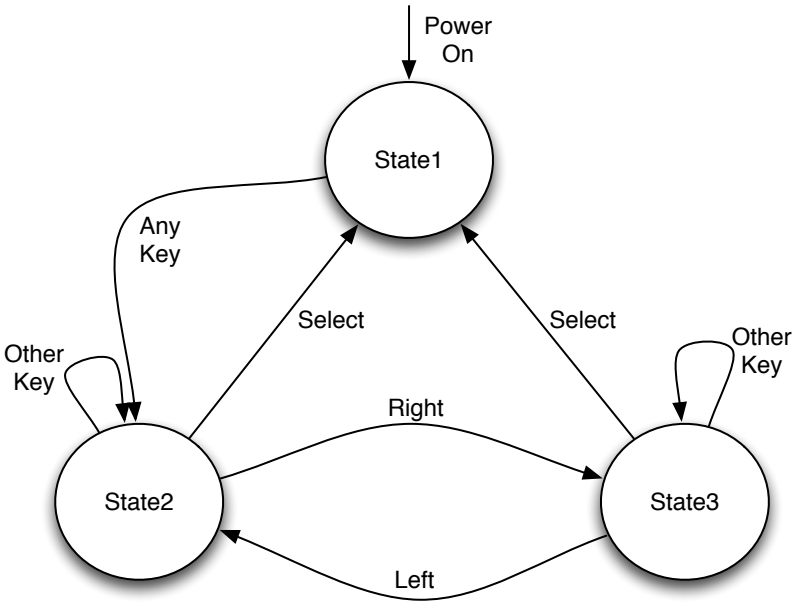

Figure 8.1: Example State Machine

The mutually tail recursive shallow Haskino code to implement this state machine is shown below, with calls to the analogKey function of Section 8.4 and a state display function which will display the state and key pressed on the LCD display, `displayState`. The top level function, `stateMachine`, calls the first mutually recursive function `state1` to provide the power on state transition. Each of the state functions follows a common pattern of displaying the state and key, waiting for the next key press, then calling one of the recursive functions to move to the next state.

```
stateMachine :: LCD -> Arduino ()
stateMachine lcd = state1 $ keyValue KeyNone
  where
    state1 :: Word8 -> Arduino ()
    state1 k = do
       displayState lcd 1 k
       key <- analogKey
       case key of
          _ -> state2 key

    state2 :: Word8 -> Arduino ()
    state2 k = do
       displayState lcd 2 k
       key <- analogKey
       case key of
          1 -> state3 key
          5 -> state1 key
          _ -> state2 key

    state3 :: Word8 -> Arduino ()
    state3 k = do
       displayState lcd 3 k
       key <- analogKey
       case key of
          2 -> state2 key
          5 -> state1 key
          _ -> state3 key
```

To transform these mutually recursive functions into an iterative function, as before the functions are first transformed from shallow to deep. Then, as was done with the simple even/odd example in the last section, a common function for all of the state functions is created (in this case `state1_deep_mut`), and the original state functions are written in terms of that function.

The common function calls the Haskino `iterateE` function, which has been modified from the original version to add an `Expr Int` parameter, similar to the `Int` parameter that was added to the `iterLoop` function in Section 8.5.

```
stateMachine_deep :: LCD -> Arduino (Expr ())
stateMachine_deep lcd = state1_deep (lit (keyValue KeyNone))
  where
    state1_deep :: Expr Word8 -> Arduino (Expr ())
    state1_deep k = state1_deep_mut (lit 0) k

    state2_deep :: Expr Word8 -> Arduino (Expr ())
    state2_deep k = state1_deep_mut (lit 1) k
```

```
state3_deep :: Expr Word8 -> Arduino (Expr ())
state3_deep k = state1_deep_mut (lit 2) k

state1_deep_mut :: Expr Int -> Expr Word8 -> Arduino (Expr ())
state1_deep_mut = iterateE i k state1_dep_mut_step
```

Finally, the step function, `state1_deep_mut_step`, is built up from the transformed bodies of the original mutually recursive state functions. The bodies are transformed using the methods described in Section 8.2. They are then inserted into a sequence of conditionals which test the step function's index parameter, `i`, determining which of the transformed bodies is executed by the step function.

```
state1_deep_mut_step :: Expr Int -> Expr Word8 ->
                                    Arduino (ExprEither Word8 ())
state1_deep_mut_step i k =
    ifThenElseEither (i ==* (lit 0))
        (do
            displayState_deep lcd (lit 1) k
            key <- analogKey_deep
            ExprLeft (lit 1) key)
        (ifThenElseEither (i ==* (lit 1))
            (do
                displayState_deep lcd (lit 2) k
                key <- analogKey_deep
                (ifThenElseE (key ==* 1)
                    (ExprLeft (lit 2) key)
                    (ifThenElseE (key ==* 5)
                        (ExprLeft (lit 0) key)
                        (ExprLeft (lit 1) key))))
            (do
                displayState_deep lcd (lit 3) k
                key <- analogKey_deep
                (ifThenElseE (key ==* 2)
                    (ExprLeft (lit 1) key)
                    (ifThenElseE (key ==* 5)
                        (ExprLeft (lit 0) key)
                        (ExprLeft (lit 2) key))))))
```

With this last step, the mutually recursive functions have been transformed into an iterative structure which may be easily transformed into C code and executed on the resource limited Arduino.

8.7 Recursion Translation with Multiple Arguments

The recursion transformations described in the examples in this chapter, and those currently implemented by the plugin described in Chapter 9 are limited in the types of functions they can transform. In the case of both self and mutually tail recursive functions, the transforms are only defined for recursive functions with zero or one arguments. Additionally, for mutually tail recursive functionals, all of the functions must have the same type signature.

However, both of these sets of limitations could be removed by modifying the transformations to operate using tuples. This would require adding tuples to the `Expr` language, which is not a small effort, but once this was done the transformations could be extended. The extensions described below make use of functions which build tuples in the `Expr` language with types defined as follows:

```
exprTuple2 :: Expr a -> Expr b -> Expr (a,b)
exprTuple3 :: Expr a -> Expr b -> Expr c -> Expr (a,b,c)
exprTuple2 :: Expr a -> Expr b -> Expr c -> Expr d -> Expr (a,b,c,d)
```

The following is an example of a recursive deep Haskino function with 2 arguments:

```
multiFunc :: Expr Word8 -> Expr Float -> Arduino (Expr Word16)
multiFunc a b = do
        ⋮
        multiFunc c d
```

The transformations in this chapter could be extended to apply to functions of this type by passing a tuple type as the initialization parameter, and including a tuple in the ExprEither type that is returned by the step function. Applying this method to our 2 argument example, we would use the following types of wrapper and step functions.

```
multiFuncE :: Expr Word8 -> Expr Float -> Arduino (Expr Word16)
multiFuncE a b = iterateE 0 (exprTuple2 a b) multiFuncEI

multiFuncEI :: Expr (Word8, Float) ->
               Arduino (ExprEither (Word8, Float) Word16)
```

We could apply a similar technique to mutually tail recursive functions which have different argument types. For example, consider the following example functions, one of which takes an 8 bit integer argument, while the other takes a float as an argument.

```
mutFunc1 :: Expr Word8 -> Arduino (Expr ())
mutFunc1 a = do
        ⋮
        mutFunc2 b'

mutFunc2 :: Expr Float -> Arduino (Expr ())
mutFunc2 b = do
        ⋮
        mutFunc1 a'
```

In this case, the transformations could be extended as shown below to once again pass `Expr` tuples as initialization arguments, with the unused elements of the tuple set to default values of zero. Only one element of the tuple would be required for each of the transformed recursive function bodies, but combined into the iterative step function the tuple type would be required.

```
mutFunc1 :: Expr Word8 -> Arduino (Expr ())
mutFunc1 a = mutFunc12 0 (exprTuple2 a (lit 0.0))

mutFunc2 :: Expr Float -> Arduino (Expr ())
mutFunc2 b = mutFunct12 1 (exprTuple2 (lit 0) b)

mutFunc12 :: Expr Int -> Expr (Word8, Float) -> Arduino (Expr ())
mutFunc12 i init = iterateE i init mutFunc12I

mutFunct12I :: Expr Int -> Expr (Word8, Float) ->
               Arduino (ExprEither (Word8, Float) Word16)
```

As the number of parameters to the recursive functions grow, higher order tuples would be required, so the size of tuple supported by the `Expr` language would be the new limit on the type of recursive function transformations which could be handled.

Chapter 9

Plugin Architecture and Implementation

The shallow to deep transformations, as well as the recursive transformations in my system, are implemented using the GHC Plugins mechanism (GHC Team, 2016). The Haskino plugin manipulates the Haskell module being compiled through a series of Core to Core passes, where Core is GHC's intermediate language(Peyton Jones & Santos, 1998).

A Haskell module's top level ModGuts data structure is carried throughout all phases of the compiler, including plugin passes. This data structure contains not only the Core of the module under compilation, but also a global reader environment of all in-scope symbols, GHC transformation rules, information about other modules imported to the one under compilation, and other information useful to the compiler pass. Each pass of a GHC plugin is defined as the following type:

```
ModGuts -> CoreM ModGuts
```

Each of the passes in our plugin transforms the list of Core Bind's, which are part of the ModGuts data type, into another list of Core Bind's in the returned ModGuts. The plugin operates on Bind's that are of the type of the DSL's Monad and Expr types, which in my case study is one of the types `(ExprB a => Arduino a)` or `(ExprB a => a)`. By the time that the compiler has translated native Haskell into Core, Bind's are separated into recursive (`Rec`) and non-recursive (`NonRec`) Bind's. The passes associated with the shallow to deep transformation operate on both Bind's constructed with `NonRec` and those constructed with `Rec`, while those associated with the tail recursion transformation operate only on the Bind's constructed with `Rec`.

The plugin has been designed to be customized for other monadic EDSLs, and not to be used just for my case study EDSL of Haskino. The types of the DSL monad and expression types are

specified in a single module as Template Haskell names, allowing the plugin to be customized quickly for a new EDSL based on the remote monad monadic structure. Similarly, tables of EDSL primitives and rules components are used in several of the passes to allow for EDSL customization, and they will be described in more detail later in this chapter. Finally, as the plugin has been developed, I have built up the basis for a plugin toolkit which is designed to be lighter weight than such tools as Hermit (Farmer et al., 2015). For example, I have generalized and made into a utility the routines from Hermit which are used to look up Core dictionaries, which frequently need to be generated as part of the transformation process.

The plugin operates on a per module basis, transforming all functions of the DSL type present in the module. GHC plugins may be invoked by specifying the `-fplugin` flag with the plugin name either on the command line, or in a compiler directive within the file. This allows us to specify on a module by module basis if the transformations will be performed. Using this method, a file may still be written directly in the Deep EDSL without transformations, or in the Shallow EDSL with transformations. The can be useful for regression testing during development of the plugin, allowing the results of the transformation to be compared the native Deep code.

The structure of the passes implemented by the GHC plugin is shown in Figure 9.1. The plugin passes are inserted in the pass chain before the standard GHC passes. Each of the passes of the plugin are described in one the following sections. The current ordering of the passes is required for the passes to function as written, and has been chosen to optimize the amount of code required for each pass. However, the optimal ordering as well as other optional orders, are still under investigation as part of ongoing research.

9.1 Simplifier Pass

The first pass ran by the plugin is a pass to execute the GHC simplifier, without any inlining, rule rewriting, or eta-expansion.

This pass is ran to complete any inlining of functions that may be have been done in the GHC compiler before it passes the Core for the module to our plugin. I found that some functions that

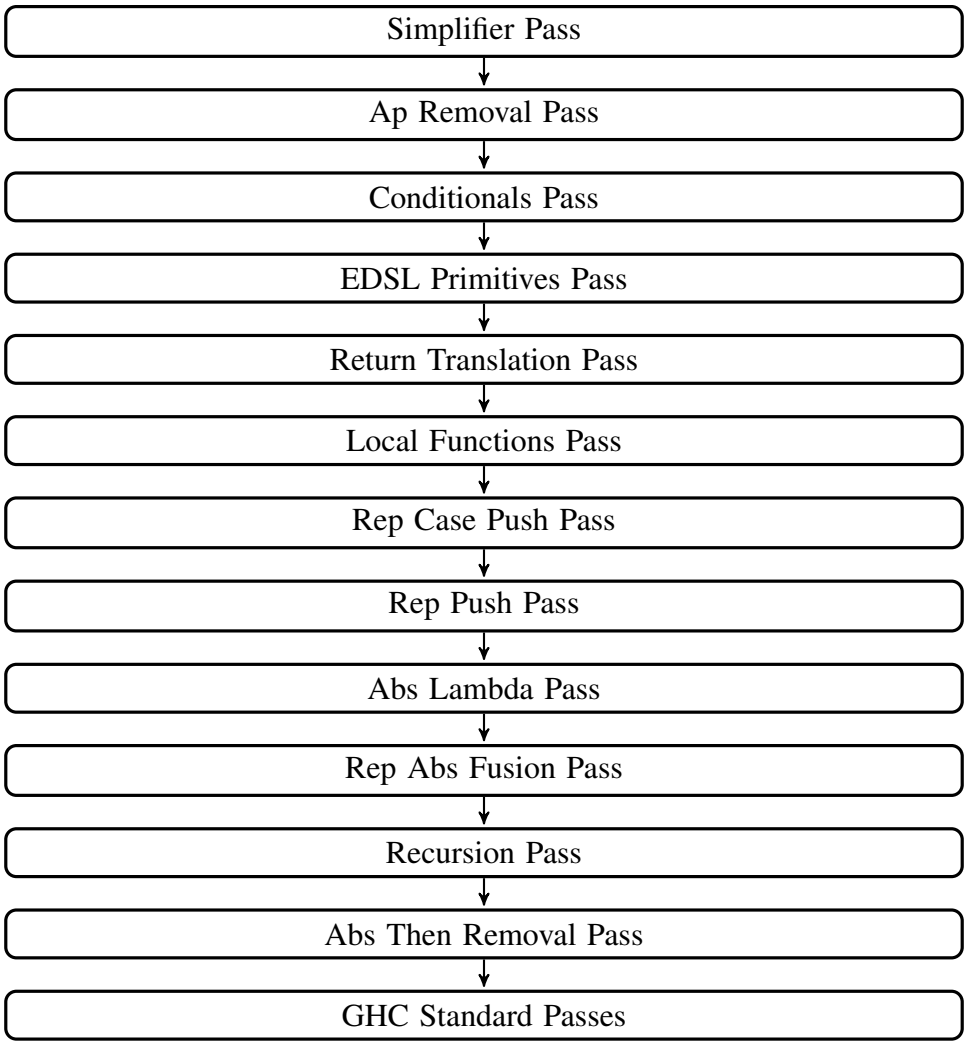

Figure 9.1: Structure of Transformation Plugin Passes

were inlined would be left in the form of `(\x -> F[x])(y)`, and running the simplifier pass will perform the function application, and leave the Core in a standard form for transformation by the rest of the plugin. The function used to call the simplifier pass is shown below.

```
simplPass = CoreDoSimplify 1 SimplMode {
             sm_names = [],
             sm_phase = Phase 2,
             sm_rules = False,
             sm_inline = False,
             sm_case_case = False,
             sm_eta_expand = False
            }
```

9.2 Ap Removal Pass

The second simplifier executed by the plugin is a pass which removes the Haskell application operator $, which is still present in the Core that the plugin receives. Replacing this operator with a standard function application reduces the number of rules that are required for subsequent passes, and the replacement is the equivelant of the following rule:

```
forall (f :: a -> b, g :: a)
  f $ g
    =
  f g
```

This simple pass provides an opportunity to demonstrate the basic methods used for performing many of the passes in the plugin. First, a new monad is defined by adding the Reader monad transformer to the compiler's base CoreM monad, to allow the ModGuts structure to be accessed from within the plugin. Some of the passes use the State monad transformer instead of the Reader, as they require state information to be updated during the pass.

```
data BindEnv = BindEnv
               { pluginModGuts :: ModGuts
               }

newtype BindM a = BindM { runBindM :: ReaderT BindEnv CoreM a }
    deriving (Functor, Applicative, Monad
             ,MonadIO, MonadReader BindEnv)
```

The top level function for this pass is `apRemovePass`, and it uses the utility function provided by the compiler, `bindsOnlyPass`, to map a transformation function, `apRemoveBind`, over each of the bindings defined in the module under compilation.

```
apRemovePass :: ModGuts -> CoreM ModGuts
apRemovePass guts = do
    bindsOnlyPass (\ x ->
        (runReaderT (runBindM $
            (mapM apRemoveBind) x) (BindEnv guts))) guts
```

In the case of this pass, the transformation function, `apRemoveBind`, calls a function, `apRemoveExpr`, for each of the expressions present in the bind. For recursive binds, a helper

function, `apRemoveBind'` is used to call `apRemoveExpr` for each of the expression in the list of recursive binds.

```
apRemoveBind :: CoreBind -> BindM CoreBind
apRemoveBind (NonRec b e) = do
  e' <- apRemoveExpr e
  return (NonRec b e')
apRemoveBind (Rec bs) = do
  bs' <- apRemoveExpr' bs
  return $ Rec bs'
```

The expression transformation function, `apRemoveExpr`, is a recursive function which handles each of the constructors present in a CoreExpr. The segment below contains only two of the possible constructors, the rest were omitted for brevity.

```
1  apRemoveExpr :: CoreExpr -> BindM CoreExpr
2  apRemoveExpr e = do
3    apId <- thNameToId '($)
4    case e of
5            ⋮
6      App e1 e2 -> do
7        let (f, args) = collectArgs e
8        let defaultReturn = do
9                e1' <- apRemoveExpr e1
10               e2' <- apRemoveExpr e2
11               return $ App e1' e2'
12       -- Pattern match instances of:
13       -- <$> :$ Type t1 :$ Type t2 :$ dict :$ f :$ g
14       -- and replace with f :$ g
15       case f of
16           Var fv | fv == apId -> do
17               case args of
18                   [_, _, _, f', arg] -> do
19                       arg' <- apRemoveExpr arg
20                       return $ mkCoreApps f' [arg']
21                   _ -> defaultReturn
22           _ -> defaultReturn
23     Lam tb el -> do
24       e' <- apRemoveExpr el
25       return $ Lam tb e'
26           ⋮
```

The `thNameToId` function on line 3 of the function is a utility function that converts Template Haskell names into GHC ID's. In the App case of the pattern match (line 6), the function searches

for the ID desired ($) in the first expression of the App. If it is found, it is replaced with a direct function application, after first recursively calling the apRemoteExpr on it's argument. The other case of the pattern match shown here, the Lam constructor (line 23), is shown to demonstrate that we need to call apRemoteExpr recursively for all of the other instances of CoreExpr that may be present.

9.3 Conditionals Pass

The Conditionals Pass transforms standard Haskell if-then-else expressions into the DSL's embedded if-then-else constructs. The pass searches the Core for two alternative Case expressions with alternattives of False and True, which return a type of Arduino a, and transforms them into the EDSL's IfThenElseE primitives according to the methods in Section 7.2. Note, that this transformation will also transform the syntax of Haskell Case pattern matching with guards of the type shown in the analogKey example in Section 8.4.

The transformation performs the equivalent of the rule that was presented in Section 7.2

```
forall (b :: Bool) (m1 :: ExprB a => Arduino a)
                   (m2 :: ExprB a => Arduino a).
if b then m1 else m2
  =
abs <$> ifThenElseE (rep b) (rep <$> m1)
                            (rep <$> m2)
```

Just like the method described for the Ap Removal pass, this pass performs a per bind transformation, as well as a recursive transformation function over the CoreExpr present in the binds. When a Case constructor is found that returns a value of the Arduino monad type, and has two alternatives, one False and the other True, it calls the function condTransform to perform the transformation. condTransform is called with 3 arguments, the type of the case statement, the Case expression (which is the boolean expression of the if-then-else), and the list of alternatives to the Case, after recursively transforming the Case expression and the alternatives.

The condTransform function is shown below to note a few of the features that are used throughout the plugin code. In line 6, we use the GHC utility function tyConAppArgs to retrieve

the base type of the Case. For example, if the Case is of the `Arduino Word8` type, this will return the `Word8` type. This base type is needed to pass as a parameter to the IfThenElseE constructor on line 17, where the GHC utility function `mkCoreApps` is used to construct the constructor application. The other parameter needed to construct the IfThenElseE is the dictionary, `condDict`, as IfThenElseE is a member of the ArduinoConditional typeclass, and uses ad-hoc polymorphism.

```
1  condTransform :: Type -> CoreExpr -> [GhcPlugins.Alt CoreBndr] ->
2                   CondM CoreExpr
3  condTransform ty e alts = do
4    case alts of
5      [(_, _, e1),(_, _, e2)] -> do
6        let [ty'] = tyConAppArgs ty
7
8        ifThenElseId <- thNameToId ifThenElseNameTH
9        condDict <- thNameTyToDict monadCondTyConTH ty'
10
11       -- Build the args to ifThenElseE
12       arg1 <- repExpr e
13       e1' <- fmapRepBindReturn e1
14       e2' <- fmapRepBindReturn e2
15
16       -- Build the ifThenElse Expr
17       let ifteExpr = mkCoreApps (Var ifThenElseId)
18                                 [Type ty', condDict, arg1, e2', e1']
19
20       -- Apply fmap of abs
21       tyCon <- thNameToTyCon monadTyConTH
22       fmapAbsExpr (mkTyConTy tyCon) ty' ifteExpr
23     _ -> return e
```

This dictionary is found on line 9, where our utility function `thNameTyToDict` is used to lookup the dictionary, given the Template Haskell name of the typeclass, and the type. The dictionary utility functions are derived from functions developed for use in Hermit (Farmer et al., 2015). Throughout the plugin, typeclass and type names are abstracted to use general names, not the specific ones. In this example `ifThenElseNameTH` and `monadCondTyConTH` are used instead of the Haskino specific `ifThenElseE` and `ArduinoConditional`. This will allow reuse of the plugin with a monadic EDSL other than Haskino, by defining the abstract names appropriately for the specific monad in use.

This pass also handles the transformation of the other type of conditional in Haskino, a conditional over the expression language, `Expr`. In this instance, similar to the monadic conditional, the plugin looks for two alternative Case expressions with alternative of False and True, which return a type in the EDSL's expression type class, `ExprB`, and transforms them into the EDSL's `ifB` primitive.

9.4 EDSL Primitives Pass

The EDSL Primitives Pass translates the EDSL primitives (in the case of a Remote Monad based EDSL, commands and procedures are the primitive data types) from their shallow form to their deep form, as was shown by the rules in Section 7.1.

Like the other passes so far, this pass performs the equivalent of the rules using a per bind transformation function. The primitives to be translated are specified by a table of pairs of identifiers, the first element of the pair being the shallow version of the primitive, and the second element being the deep version of the primitive.

```
data XlatEntry = XlatEntry {  fromId   :: BindM Id
                            , toId     :: BindM Id
                           }

xlatList :: [XlatEntry]
xlatList = [  XlatEntry (thNameToId 'System.Hardware.Haskino.setPinMode)
                        (thNameToId 'System.Hardware.Haskino.setPinModeE)
            , XlatEntry (thNameToId 'System.Hardware.Haskino.digitalWrite)
                        (thNameToId 'System.Hardware.Haskino.digitalWriteE)
            ⋮
```

The transformation function recursively searches the Core for function applications of one of the first elements in the `xlatList`, and replaces it with an application of the second element. The function needs to search both `App` and `Var` constructors in the CoreExpr, as primitives without any parameters will show up as expressions in a `Var` constructor.

In addition to simply substituting the primitives, shallow for deep, the pass compares the types of the two versions of the primitive, and performs the translation by adding application of `rep`

and `abs` functions as needed. It first either applies a `rep` function to each of the arguments of the primitive, using either simple function application or a fmap function, depending on if the argument is monadic or not. It also applies an `abs` to the return value of the primitive using the fmap function, since the return is of a monadic type.

9.5 Return Translation Pass

For monadic based DSLs, such as Haskino, instances of `return` functions need to be transformed just as the EDSL primitives are. This pass transforms the returns with the equivalent of the following rule:

```
forall (x :: ExprB a => a)
  return x
    =
  abs <$> return (rep x)
```

This transformation is equivalent to the transformation of procedures used in the EDSL Primitives Pass.

9.6 Local Functions Pass

As the plugin was being designed, there were two options for handling local function definitions in the module being compiled. As running most deeply embedded DSLs consists of inlining those functions, the first option was to simply inline any applications of those functions in other functions inside of the module in this pass. However, it was planned to add Lambda expressions to the Haskino Deep EDSL in the next phase of research, and therefore a method that did not inline everything in this pass of the transformation was desired. In addition, Haskino was intended to work with programs that spanned Haskell modules, and not just be limited to a single module, so the transformations also needed to work with programs that span multiple modules. So, instead of simply inlining all of the local functions, the decision was made to transform the types of local functions within this pass of the transformation.

In doing this transformation, this pass replaces the shallow function body with a call of a deep version of the function, as in the following example:

```
myRead :: Word8 -> Arduino Bool
myRead p = abs <$> (myReadDeep (rep p))

myRead_deep' :: Expr Word8 -> Arduino Bool
```

What was the former body of the shallow version is transformed by this pass to become the new body of the deep version. Using this method, a module being transformed that imports a previously transformed module will pass type checking during the initial GHC type checking phase, before the untransformed Core is given to the plugin, since both shallow and deep versions will be present in the previous module.

This pass also replaces applications of shallow functions, with applications of the deep functions. This means it applies a `rep` function to each of the arguments of the function, and an `abs` to the return value of the function, similar to how EDSL primitives are transformed.

This pass is more complicated that the passes we have seen so far. It performs two sequences of transformations over the binds in the module, one to handle the function type signature transformation, and one to handle the function application site changes. In the first transformation sequence, it examines each bind, and choses the ones where the return value of the bind is one of two types, either `(exprB a => Arudino a)`, or `(exprB a => a)`, as these are the two types of functions which are transformed from shallow to deep by this pass. For each of those functions, a new deep GHC id is created with a name that is the name of the shallow function appended with the string "_deep". This new id is given a function type that is created by taking the application types and return types of the shallow function, and transforming them to `Expr` types. It should be noted here that only arguments whose type is a member of the `ExprB` type class are transformed. Other argument types are left unchanged, and calculations on those arguments will not be deeply embedded, but will be performed on the host. A new shallow body expression is then created, which is written as a call of the deep function as shown in the example earlier in this section. The original shallow body is then transformed, to become the new deep body, applying a `rep` function to it to change it

to the deep type. Since the type of the arguments of the function have been changed to make them deep, the body must also be transformed by replacing any occurrences of the arguments of `ExprB` types in the body with `abs` applied to the argument. At this point the original bind is replaced with the two new binds, the shallow id bound to the new shallow body written in terms of the deep id, and the new deep bind id bound to the transformed shallow body.

In addition to replacing the original bind in the list of top level binds for the module, these two new binds are also placed in a dictionary, with the shallow id as the key to the new dictionary entry, and the deep id as the dictionary entry value. Once this dictionary of shallow to deep bind id's is created, the list of id's which are exported from the module is updated in the modGuts structure as well. If the shallow id from the dictionary exists in the list of binds which is exported, then we add the deep id to the export list, so that other modules importing this one may use our newly transformed bind.

At this point, we are ready to start the second transformation sequence through top level binds. In this transformation sequence, the expressions associated with each bind are recursively transformed using the dictionary that was created in the last sequence. There is one complication to this procedure though, and that is the handling of let expressions found within each of the top level binds. As the bind expression is recursively transversed, these let bindings are transformed in the same manner as the top level binds were. The dictionary of shallow to deep transformations then becomes a stack of dictionaries, one entry in the stack for each level of let binding. During the recursive transversal of the expression, when a new transformable let binding is found, a new dictionary is pushed to the stack containing the shallow to deep id mapping for that let, and when the let binding is exited, the dictionary is popped from the stack.

During the same recursive transversal of the expressions, any function applications with either `(exprB a => Arudino a)`, or `(exprB a => a)` return types are examined. There are four possible choices on what is done with these function applications.

1. The function is checked against a list of the deep primitives. If the function is in this list, then it is already of a deep type, and is not transformed.

2. If the function is an element of any of the dictionaries on the stack, then the function is replaced by the deep id found in the dictionary for the shallow id key, adjusting it's arguments and returns values with `rep` and `abs` applications as we did in the primitive pass.

3. If the function is not found in either the primitive list or the dictionary, then it is assumed to be imported from another module. An id with the name of the function, appended with the string "_deep", is searched for in the global reader environment that is part of the ModGuts structure. The deep id returned is then substituted for the shallow function, and it's arguments and return value adjusted as with the primitive pass.

4. If the shallow id is not found in the global search, it indicates an error with the plugin implementation, and a compiler error message is issued.

9.7 Rep Case Push Pass

The Rep Case Push pass is the first of the passes used to manipulate the worker-wrapper functions which were inserted by the previous passes, transforming shallow expression functions to deep, and moving the worker-wrapper functions for possible fusion in the later passes.

This pass was not required by the original simple examples that were used to test the plugin. However, when more complicated Haskino code was used with the plugin, such as the Haskino LCD library, it was found that the plugin did not properly translate all Haskino code. For example, the Haskino LCD library defines a `LCDController` type as follows:

```
data LCDController =
    Hitachi44780 {     lcdRS       :: Pin
                       , lcdEN       :: Pin
                       , lcdD4       :: Pin
                       , lcdD5       :: Pin
                       , lcdD6       :: Pin
                       , lcdD7       :: Pin
                       , lcdBL       :: Maybe Pin
                       , lcdRows     :: Word8
                       , lcdCols     :: Word8
                       , dotMode5x10 :: Bool }
    | I2CHitachi44780 {  address     :: Word8
                       , lcdRows     :: Word8
                       , lcdCols     :: Word8
                       , dotMode5x10 :: Bool }
```

This type allows the user to define an instance of a LCD controller that is attached to their Arduino board, and specify how it is attached (either by a direct, multi-pin connection, or using an I2C device), what pins it is attached to, and what it's capabilities are.

Since we have not defined an explicit Case structure in the Arduino EDSL, it was expected no instances of Core `Case` constructors would be found at this stage of the plugin pipeline, as they would have been eliminated by the Conditionals Pass described in Section 9.3. However, this was found to not be the case. The accessor function, `lcdCols` that is defined by Haskell to retrieve the lcdCols element of the data structure will be generated as the following Core code, and it contains a case that will not be eliminated.

```
lcdCols :: LCDController -> Word8
 lcdCols =
  \ (ds_d6T9 :: LCDController) ->
    case ds_d6T9 of _ [Occ=Dead] {
      Hitachi44780 ds_d6Ta ds_d6Tb ds_d6Tc ds_d6Td ds_d6Te ds_d6Tf
                   ds_d6Tg ds_d6Th ds_d6Ti ds_d6Tj ->
        ds_d6Ti;
      I2CHitachi44780 ds_d6Tk ds_d6Tl ds_d6Tm ds_d6Tn -> ds_d6Tm
    },
```

When this is translated to the deep version, by the Local Functions Pass described in Section 9.6, the following Core will then be the output of that phase.

```
lcdCols :: LCDController -> Expr Word8
lcdCols_deep' =
  \ (ds_d6T9 :: LCDController) ->
    rep_
      @ Word8
      System.Hardware.Haskino.Expr.$fExprBWord8
      (case ds_d6T9 of _ [Occ=Dead] {
         Hitachi44780 ds_d6Ta ds_d6Tb ds_d6Tc ds_d6Td ds_d6Te ds_d6Tf
                      ds_d6Tg ds_d6Th ds_d6Ti ds_d6Tj ->
           ds_d6Ti;
         I2CHitachi44780 ds_d6Tk ds_d6Tl ds_d6Tm ds_d6Tn -> ds_d6Tm
       }),
```

From this Core output, we can see that the `rep` application is not where it needs to be to transform the return value of the accessor function to the deep type. To handle this case, as well as others that appear when using this type of data type in Haskino code, we need to define yet another type of transformation rule to manipulate the `rep` worker-wrapper function.

The Rep Case Push pass is the plugin pass that implements this additional transformation. It performs the equivalent of the following rule, moving the rep application through the case, and applying it to each of the constituent alternatives of the case.

```
forall (a1 :: ExprB a => a) ... (an :: ExprB a => a)
rep (case c of e1 -> a1 ... cn -> an)
    =
case c of e1 -> rep_ a1 ... cn -> rep_ an
```

Like many of the other passes we have discussed, this pass performs a simple recursive search of the Core constructors, looking for an instance of the `rep` function applied to a `Case` constructor which returns a type of the `ExprB` typeclass. It then replaces that with a Case constructor with the `rep` function applied to each of the alternatives.

Going back to our example in this section, after this pass has transformed the core for the `lcdCols` accessor function, it looks like the following, with the `rep` application now in the proper position.

```
lcdCols_deep' :: LCDController -> Expr Word8
lcdCols_deep' =
   \ (ds_d6T9 :: LCDController) ->
     case ds_d6T9 of _ [Occ=Dead] {
       Hitachi44780 ds_d6Ta ds_d6Tb ds_d6Tc ds_d6Td ds_d6Te ds_d6Tf
                    ds_d6Tg ds_d6Th ds_d6Ti ds_d6Tj ->
         rep_
           @ Word8
           System.Hardware.Haskino.Expr.$fExprBWord8
           ds_d6Ti;
       I2CHitachi44780 ds_d6Tk ds_d6Tl ds_d6Tm ds_d6Tn ->
         rep_
           @ Word8
           System.Hardware.Haskino.Expr.$fExprBWord8
           ds_d6Tm
     }
```

9.8 Rep Push Pass

The Rep Push pass is the second of the passes used to manipulate the worker-wrapper functions which were inserted by the previous passes. This pass is performed in the plugin in a similar method to the EDSL primitive pass. It also contains a table of pairs of the functions with the from and to functions to transform for each of the EDSL Expr language operations. (In the example in Section 7.1 discussing the Rep Push transformations, this pair consists of the `(||)` and `(||*)` functions). One of the sets of pairs is defined for each of the DSL's Expr operations, and is used to move the operation from a shallow operation in basic Haskell types, to a deep operation using the Expr language operators.

```
data XlatEntry = XlatEntry {  fromId         :: BindM Id
                            , toId           :: BindM Id
                           }
xlatList :: [XlatEntry]
xlatList = [  XlatEntry (thNameToId 'not)
                        (thNameToId 'Data.Boolean.notB)
            , XlatEntry (thNameToId '(||))
                        (thNameToId '(||*))
                ⋮
```

As with the EDSL Primitive pass, the transformation function recursively searches the Core for function applications of one of the first elements in the `xlatList`, and replaces it with an application of the second element, applying a `rep` to each of the arguments of the replacement function. It also compares the types of the original and the replacement functions, and automatically generates the required type arguments and dictionaries for the functions.

While designing the type comparison functions for the operator transformation, a difficulty was encountered. When attempting to translate shallow comparison operators such as `(==)` to a deep comparison operator, `(==*)`, it was found that the translation needed to deal not just with type arguments and dictionaries, but also with coercions. This is due to the fact that the definition of the EqB, OrdB, and IfB typeclasses have function types which contain type equality constraints in their definitions, as show below with the `EqB` class.

```
class Boolean (BooleanOf a) => EqB a where
  (==*), (/=*) :: (bool ~ BooleanOf a) => a -> a -> bool
```

These type equality constraints translate to coercions in the Core language. Dealing with coercions in the type comparisson and translation functions would significantly complicate the translation. Instead, it was determined to be cleaner to eliminate the type equality requirements by extending the `ExprB` typeclass to include wrapper functions for equality, ordinality, and conditional expression functions. These are the eqE, lessE (and related functions), and ifBE functions shown in the type class definition and example instance below.

```
class ExprB a where
    lit      :: a -> Expr a
    remBind  :: Int -> Expr a
    showE    :: Expr a -> Expr [Word8]
    lessE    :: Expr a -> Expr a -> Expr Bool
    lesseqE  :: Expr a -> Expr a -> Expr Bool
    -# INLINE lesseqE #-
    lesseqE a b = notB (lessE b a)
    greatE   :: Expr a -> Expr a -> Expr Bool
    -# INLINE greatE #-
    greatE a b = lessE b a
    greateqE :: Expr a -> Expr a -> Expr Bool
    -# INLINE greateqE #-
    greateqE a b = notB (lessE a b)
    eqE      :: Expr a -> Expr a -> Expr Bool
    neqE     :: Expr a -> Expr a -> Expr Bool
    -# INLINE neqE #-
    neqE a b = notB (eqE a b)
    ifBE     :: Expr Bool -> Expr a -> Expr a -> Expr a

instance ExprB Word8 where
    lit = LitW8
    remBind = RemBindW8
    showE = ShowW8
    -# INLINE lessE #-
    lessE = (B.<*)
    -# INLINE eqE #-
    eqE = (==*)
    -# INLINE ifBE #-
    ifBE = ifB
```

Adding these functions to the typeclass eliminates the coercions from the Core and simplifies the translation, with a small increase to complexity of the EDSL. Also, default definitions were added for the non-required functions in the EqB and OrdB typeclasses (such as `lesseqE` and `neqE`) to provide convient translation targets for the translation table. The additional functions in the typeclass are marked as `INLINE` in their definitions so as to not add an additional function application in the generated code.

9.9 Abs Lambda Pass

The Abs Lambda pass is the last of the passes used to manipulate the worker-wrapper functions in preparation for fusion. This pass performs the equivalent of two of the rules described in Section 7.1. The first of these two rule analogues pushes the `abs` function applications through monadic >>= and >> operators.

```
forall (f :: ExprB a => Arduino a)
  (g :: ExprB a,b => a -> Arduino (Expr b))
  (k :: ExprB b, c => b -> Arduino c).
    (f >>= (abs_ <$> g)) >>= k
       =
    (f >>= g) >>= k . abs_
```

The second rule, which moves the `abs` inside of lambdas, which has the form shown below.

```
forall (f :: Arduino a).
(\ x -> f[x]) . abs
  =
(\ x' -> let x=abs x' in f[x])
```

In the implementation of this pass, these two rules are applied in one recursive traversal through the Core code. The pass identifies bind chains where the `abs` function applications need to be moved to the end of the bind chain. When the end of the bind chain is also found to be a lambda, the abs is eliminated, the lambda argument x is renamed to x_abs and it's type is changed to Expr a. As a final step, any occurrence of x in the body of the lambda is replaced with abs(x_abs).

9.10 Rep Abs Fusion Pass

This pass fuses the `rep` and `abs` pairs that have been moved next to each other by the previous passes. It performs the equivalent of the two rep-abs fusion rules described in Section 7.1.

```
forall x.
rep(abs(x))
  =
x
```

```
forall m.
rep <$> (abs <$> m)
  =
m
```

After this pass is complete, there will be some applications of `rep` left in the Core, where it is required to lift literal basic Haskell values into the EDSL's `Expr` language.

9.11 Recursion Pass

The Recursion Pass transforms Deep EDSL tail recursive functions into the Deep EDSL's `iterateE` construct. It performs the equivalent of rules described in Chaper 8 with a Core-to-Core pass, and only operates on Core Bind's constructed with `Rec`.

The plugin as stands transforms tail recursive Haskino functions with zero or one arguments. Recursive functions with larger number of arguments are currently flagged to the user as not transformable. Transformation of these type of functions could be added to the pass, but would require the addition of tuples to the Haskino Deep EDSL, as described in Section 8.7.

The pass starts by examining all of the recursive binds in the module, looking for binds returning a type of the EDSL monadic type (`Arduino`) over an ExprB type. Those with zero or one arguments, where the type signature for all of the recursive functions in the recursion group have the same type signature (same argument type and same return type), are chosen for transformation. Those that do not meet this criteria are flagged as not transformable by printing a compiler error message. For those that have the proper number of arguments, it creates a new wrapper function as a non-recursive bind, and populates it with an instance of the `iterateE` function with the proper types. It then takes the former bodies of the mutually recursive function (or of the body of the single recursive function), and passes them to a function to perform the recursive transformation, the result of which then becomes the third argument to the `iterateE` function. The original recursive functions are then rewritten to call the new wrapper function, passing the recursive function index and the initialization value as arguments to the wrapper function. If the original recursive functions had no arguments, then the `iterateE` function will have a return type of

`ExprEither (Expr ()) (Expr a)`, and the initialization value will simply be `LitUnit`, which is the only value in the `Expr ()` type.

The transformation function performs the equivalent of the following three rules from Chapter 8. The first moves the Done constructor through any conditionals:

```
forall f x.
    Done <$> if b then f else g
        =
    if b then (Done <$> f) else (Done <$> g)
```

The second moves the Done constructor call to the end of the bind chain.

```
forall f g.
    Done <$> (f >>= g)
        =
    f >>= \ x -> Done <$> g x.
```

And the final, in the branches where the recursive function call is present, eliminates the recursive call, instead inserting the Step constructor.

```
forall x.
    Done <$> f_index x
        =
    (Step index) <$> (return x)
```

To implement these rules, the transformation function, `transformRecur`, walks the monadic bind chain of the function, and at each stage, performs one of five options.

1. If function at that position is an `ifThenElseE`, it is replaced with an `ifThenElseEitherE` of the appropriate type, and the transformation function is called recursively for each branch of the conditional.

2. If the function at that position is a monadic bind it:

    - Checks the left hand side of the bind to determine if it is a non-tail recursive call. If it is, it issues an error message.
    - If it is not, it then calls the transformation function recursively for the right hand side of the bind.

3. If the function is a recursive call to the function under transformation, the call is replaced with a call to a monadic return. The argument to the monadic return is then wrapped with an `ExprLeft`, to signal to the iterateE primitive that the iteration will continue for this branch.

4. If the function is a return, then this branch is a non-recursive one. The argument to the return is wrapped with an `ExprRight`, to signal to the iterateE primitive that the iteration will not continue for this branch.

5. Finally, if none of the above are true, then this is a non-recursive branch, with a call to another Haskino monadic function. Therefore, we apply an fmap of `ExprRight` to it, to indicated the return value of the Haskino function will be the return value of the overall iteration structure.

Once the bodies of the recursive functions are transformed, a new step function is created which is passed as the final parameter of the `iterateE` function in the wrapper function. The created step function consists of a scaffolding of ifThenElseEither conditionals which test the function index passed to the step function, ensuring that the proper transformed recursive function body is executed for each call of the step function. Once this scaffolding is created, the transformed bodies are placed within the conditional arms. If there was only one recursive function in the recursive group, then the conditional scaffolding will not be created, and the transformed recursive function body will be inserted directly as the step function body.

9.12 Abs Then Pass

After the other transformation passes, there may be some applications of `abs` left in the Core, but due to Haskell's lazy evaluation, these will never be evaluated. This will occur when an EDSL procedure's return value is not bound to a lambda argument, but is instead used with the >> operator instead of the >>= operator. To simplify the generated Core, these unevaluated instances of `abs` are eliminated by this pass.

9.13 Debugging the Plugin

There were several facilities used for debugging during the development of the plugin. The first of these was a pass that was defined for the plugin that dumped all of the binds in the module under compilation at the stage of the pipeline where the debug pass was inserted. It's definition is relatively simple:

```
dumpPass :: ModGuts -> CoreM ModGuts
dumpPass guts = do
  putMsg $ ppr (mg_binds guts)
  return guts
```

It uses the `putMsg` facility of the CoreMonad to output a SDoc type to the console. The `ppr` function is a pretty printer for members of the `Outputable` typeclass, which includes most of the internal data structures of GHC. This pass simply pretty prints all of the binds in the `ModGuts` structure to the console.

For debugging the inner workings of a plugin pass, the same `putMsg` function (and it's cousin, `putMsgS` which takes a String as a parameter), were used within the code of the pass itself. To enable this, the monads used in the translation passes were made instances of a generic pass typeclass, which defines a function to lift CoreMonad functions into the pass monad.

```
instance PassCoreM BindM where
    liftCoreM m = BindM $ lift m
```

Then, using this functionality, the putMsg may be used with the transformation pass code, as in this example printing out a Core structure.

```
liftCoreM $ putMsg $ ppr someCoreStruct
```

Finally, if very detailed information about the Core, including all fields and components of the internal GHC identifiers and operators, was needed, the Hackage CoreDump (Ömer Sinan Ağacan, 2015) package was modified to work with the Haskino plugin structure. This resulted in a `showPass` which may be used much like the `dumpPass`, but which prints very verbose information for detailed debugging. This detail was rarely needed, but was essential in a few cases.

9.14 Plugin Translation Limitations

The translations performed by the Haskino plugin can handle most Haskino source, however, the plugin does have limitations. The limits on recursion translation were detailed in Section 9.11. There are also known limitations to the shallow to deep translation. One of these is the inability to translate the Haskino primitive `modifyRemoteRef`. The type signatures of the shallow and deep versions of this primitive are shown below.

```
modifyRemoteRef  :: RemoteRef a -> (a -> a) -> Arduino ()
modifyRemoteRefE :: RemoteRef a -> (Expr a -> Expr a) ->
                           Arduino (Expr ())
```

The limits of the transformation are due to the inability to translate the second argument of the primitive. The translation is currently unable to handle the transformation of primitive arguments which are functions. The issue may be worked around by using a `readRemoteRef` and `writeRemoteRef` sequence. The translation could also be extended to handle the application of a `rep` to the functional argument if it is a lambda. This would require the addition of a repLambda rule similar to the absLambda rule described in Section 9.9. For a function argument that is a local bind, the translation would not need to apply a `rep`, as the local bind would already be translated to a deep function.

The second known limitation is in dealing with enumerated data types in the Haskino code. An example is from the LCD display with keyboard example described in Section 8.4. We would like to encode the key choices by deriving `Enum` on the `Key` data type, and then determine the value by doing a fromEnum as shown below.

```
data Key = KeyNone
         | KeyRight
         | KeyLeft
         | KeyUp
         | KeyDown
         | KeySelect
  deriving Enum

keyValue :: Key -> Word8
keyValue = fromEnum
```

However, this will not work with the shallow to deep translation, as there is currently not a `Expr` operation to translate from an Enum data type to an `Expr Int`. Therefore, we would need to define a new operation, `fromEnumE` as is shown below.

```
fromEnum :: Enum a => a -> Int

fromEnumE :: Enum a => a -> Expr Int
fromEnumE = lit . fromEnum
```

The translation from `fromEnum` to `fromEnumE` could be added to the Rep Push pass described in Section 9.8. However, the translation function in that pass would also need to be updated, as it is not currently able to handle the translation of operation functions that have non-`ExprB` types as parameters, as is the case with `fromEnum` and it's first parameter. Until this is updated, hand written functions which go from an enumerated data type to an integer type are required to be written in Haskino, as is the case with the key example from Section 8.4, shown below.

```
keyValue :: Key -> Word8
keyValue KeyNone   = 0
keyValue KeyRight  = 1
keyValue KeyLeft   = 2
keyValue KeyUp     = 3
keyValue KeyDown   = 4
keyValue KeySelect = 5
```

The final known limitation of Haskino's plugin translation has to do with how construction of lists of type `[Word8]` are coded in a Haskino application. We will use an example from the Haskino interpreter bootstrap case study, where we would like to recursively call the `readFrame'` function with the input list appended with the checksum. As Haskell programmer would normally write this as:

```
readFrame' $ l ++ [c]
```

However, this fails to translate. This is due to the fact GHC normally uses the `build` list constructor to construct lists. The plugin is not currently able to translate the `build` function. Allowing this syntax would require adding a plugin translation pass that either applies a rule for moving `rep` functions inside of the build function, transforming it's parameters from `[Word8]` to

`Expr [Word8]`, or translates the build function application into a combination of `cons` operations and an empty list constructor. For now, Haskino code using list construction must construct the lists with a `cons` (`:`) and empty list sequence (`[]`) as shown below.

```
readFrame' $ l ++ c : []
```

Chapter 10

Case Studies

The compiler plugin described in Chapter 9, implementing the rules and algorithms detailed in earlier chapters, allows a user to write in a shallowly embedded syntax, and have the program automatically translated to a deeply embedded program, which may then be compiled to C. In this chapter, we will examine two case studies which use this system to implement larger programs than the simple ones we have examined up to this point. Although directly comparing entire programs written in Haskino, which are then compiled to C, to the same programs written directly in C is difficult, we can compare subsets of the programs.

The first case study examines writing a driver for a common Arduino peripheral, a multi-line LCD display (which was discussed in Sections 5.4.2, 9.7, and 8.6). The second case study examines writing the Haskino interpreter in Haskino itself, allowing Haskino to "bootstrap" itself.

10.1 Case Study: LCD Driver and Applications

The multi-line LCD display, a common Arduino peripheral, and snippets of code dealing with it, have been discussed in several of the previous chapters. The writing of an LCD driver for Haskino was a key part of the research, and enabled many of the examples in this dissertation.

In Section 9.7, the `LCDController` data type was discussed in relation to how it had to be handled differently by the compiler plugin. The `LCDController` data type is part of a larger datatype that describes the interface to the controller, from both a configuration and a state standpoint. The top level `LCD` data type, as well as the state related `LCDData` data type are shown below.

```
data LCD =
  LCD {
     lcdController   :: LCDController -- Actual controller
   , lcdState        :: LCDData       -- State information
   }

data LCDData =
  LCDData {
      lcdDisplayMode    :: RemoteRef Word8 -- left/right/scrolling etc.
    , lcdDisplayControl :: RemoteRef Word8 -- blink on/off, dsply on/off
    , lcdGlyphCount     :: RemoteRef Word8 -- count of custom glyphs
    , lcdBacklightState :: RemoteRef Bool
    }
```

The state data type, `LCDData` is made up of Haskino `RemoteRef`'s which are used to store state information about the display for the driver during execution. An instance of the `LCD` data type is created with the lcdRegister function, which is passed as a parameter an instance of the `LCDController` data type, specifying how the controller hardware is wired into the Arduino system. The `lcdRegister` function then creates four new `RemoteRef`'s for the new instance of the `LCDData` data type, and that is combined with the `lcdController` argument to create the overall data structure which identifies a specific instance of the hardware controller.

```
lcdRegister :: LCDController -> Arduino LCD
lcdRegister controller = do
    mode <- newRemoteRef 0
    control <- newRemoteRef 0
    count <- newRemoteRef 0
    backlight <- newRemoteRef True
    let ld = LCDData { lcdDisplayMode    = mode
                     , lcdDisplayControl = control
                     , lcdGlyphCount     = count
                     , lcdBacklightState = backlight
                     }
    let c = LCD { lcdController = controller
                , lcdState = ld
                }
    initLCD c
    return c
```

The new instance of the `LCD` data type is retured from the registration function, and is passed to all of the driver functions as the identification of the display instance, which allows an application

to use multiple displays if so desired. As the `LCD` data type is not an instance of `ExprB`, it is not translated by the compiler plugin during the shallow to deep transformation.

Two of the applications used as examples in this dissertation use the LCD Display Driver. The LCD Counter example in Section 5.4.2 is the first of these. Also, the mutual recursion state machine example in Section 8.6 uses the driver as well. The driver proved invaluable during the development of the system by providing an easy to read visual output for debugging, as well as being representative of the required interfaces for an Arduino peripheral and driver.

10.1.1 Simple LCD Application

A common first example of a programming language is a "Hello World" application. I will use a similar application as an intro example of programming the Arduino using the LCD driver. As this research was done at the University of Kansas, my example program will be called "Hello Lawrence". In the main loop, it will display each of following three strings on the LCD display: "Rock", "Chalk" and "Jayhawk". After displaying each string it will delay for 1500ms. The C language version that would be written directly with the Arduino IDE is shown below:

```
#include <LiquidCrystal.h>
const int rs = 8, en = 9, d4 = 4, d5 = 5, d6 = 6, d7 = 7;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

void setup() {
  lcd.begin(16, 2);  // Specify the size of the display
}

void loop() {
    lcd.home();
    lcd.print("Rock");
    delay(1500);
    lcd.home();
    lcd.print("Chalk");
    delay(1500);
    lcd.home();
    lcd.print("Jayhawk");
    delay(1500);
    lcd.clear();
}
```

The Arduino C code uses the Arduino provided library, LiquidCrystal, to program the LCD display. Like the Haskino LCD driver's `lcdRegister` function, this library requires creating a LiquidCrystal object which specifies which pins the LCD display is connected to.

The Haskino version of the same application is shown next:

```
hitachi :: LCDController
hitachi = Hitachi44780 { lcdRS = 8
                       , lcdEN = 9
                       , lcdD4 = 4
                       , lcdD5 = 5
                       , lcdD6 = 6
                       , lcdD7 = 7
                       , lcdBL = Just 10
                       , lcdRows = 2
                       , lcdCols = 16
                       , dotMode5x10 = False
                       }

hello :: LCD -> Arduino ()
hello lcd =
    lcdHome lcd
    lcdWrite lcd $ litString "Rock   "
    delayMillis 1500
    lcdHome lcd
    lcdWrite lcd $ litString "Chalk  "
    delayMillis 1500
    lcdHome lcd
    lcdWrite lcd $ litString "Jayhawk"
    delayMillis 1500
    lcdClear lcd
    hello lcd

helloLawrence :: Arduino ()
helloLawrence = do
    lcd <- lcdRegister hitachi
    lcdBacklightOn lcd
    hello lcd
```

This main application code for this simple application is written similarly in both languages. The library source for both LCD drivers will not be shown here for the sake of brevity, but I will use this example program to compare the resource utilization and performance of directly programming it in C using the Arduino IDE, against utilizing Haskino to write the program and generate C code from it.

10.1.2 Resource Usage Comparison

Table 10.1 shows the differences in RAM and Flash usage between the two versions, simply measured by the output of the Arduino `avr-size` tool which is run following the program linking. For the Haskino version, the static definition of the task control block was removed before compiling, as it includes the stack space for the default task allocated by the runtime, and stack size is not included in the pure C version by the `avr-size` tool. Stack usage was calculated by passing `-fstack-usage` to `gcc` during compilation, and then analyzing the call tree to find the maximum approximate static usage. The Haskino version uses the dynamic memory management described in Section 6.7, and the minimum number of allocation blocks required for the application was determined through experimentation, and those buffer sizes are reflected in the numbers in the table.

	Hello Lawrence Application in C	Hello Lawrence Application in Haskino
Flash Size	2093 bytes	5926 bytes
Static RAM Size	61 bytes	144 bytes
Stack RAM Size	24 bytes	24 bytes
Total RAM Size	85 bytes	168 bytes
Uno Flash Usage	6.4%	18.1%
Uno RAM Usage	4.1%	8.2%

Table 10.1: Summary Sizing of Hello Lawrence Application Written in C and Haskino

	Hello Lawrence Application in C	Hello Lawrence Application in Haskino
List Buffers	–	60 bytes
Remainder of Application/Libs	61 bytes	84 bytes
Total Ram Size	61 bytes	144 bytes

Table 10.2: Detail of Static RAM Usage in Hello Lawrence Application

As can be seen from the measurements in the table, the Haskino version uses more Flash and RAM than the C version does. The static RAM usage of the two versions is shown in more detail in Table 10.2. The Haskino version requires list process buffer space where the C version does not. The remaining memory difference between the two versions is for the remainder of the

applications/libraries/runtime for the Haskino version, which requires 23 more bytes than the C version, which is a 37% increase.

The flash memory usage consists of several different components, and those are detailed in Table 10.3. The Haskino Runtime makes up approximately 17% of the total flash storage requirements of the Haskino version, and this is a flat "tax" that is independent of the application size. Looking at the application and library flash storage requirements, we can see that the Haskino version requires approximately 200% more flash storage than the C version does. In section 10.1.4, we will examine the cause of that large expansion, and how it might be reduced.

	Hello Lawrence Application in C	Hello Lawrence Application in Haskino
Arduino Libraries	978 bytes	978 bytes
Haskino Runtime	—	1616 bytes
Application and LCD Library	1114 bytes	3332 bytes
Total Flash Size	2092 bytes	5926 bytes

Table 10.3: Detail of Flash Usage in Hello Lawrence Application

10.1.3 Processing Time Comparison

In addition to comparing the resource utilization for the LCD application case study, the processing time required for the main loop of the application was also compared. For both the C and Haskino versions, the time required for the main loop with the delays removed was measured. In other words, the time required to display the three strings in the message was measured. The loop was ran for 1000 iterations, and the measured time was then divided by 1000 to calculate the number in Table 10.4. The Haskino version was 2.6% faster than the C version, which may be due to the inlined nature of the Haskino generated code.

	Hello Lawrence Application in C	Hello Lawrence Application in Haskino	Delta
Main Loop time	12.65 ms	12.32 ms	-2.6%

Table 10.4: Processing Time in Hello Lawrence

10.1.4 Duplicated Code

There is a large difference between the flash memory resources used by the Haskino and C versions of the application, in fact the Haskino version uses almost three times as much flash. Analysis of the generated code reveals the large contributor to the size of the Haskino version. The structure of the application is such that it writes commands to the LCD display unit using sequences of `digitalWrite's` in many locations in the generated C code. Due to the inherent inlining nature of executing a deeply embedded EDSL, this command code is repeated many times. In the case study application there are 17 copies of the command code, which is generated from the Haskino source function `transmit`. Measuring the size of the generated code for this function reveals that it requires 140 bytes of Flash space per instance of the generated code. If this code were moved to a generated C function, 16 copies of it could be eliminated. This would require inserting 17 function calls in their place, and estimating 16 bytes of code per function call, the estimated savings would be 16 * 140 - 17 * 16, or 1968 bytes. Table 10.5 shows the differences between the interpreter version after these savings.

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino
Arduino Libraries	978 bytes	978 bytes
Haskino Runtime	—	1616 bytes
Application and LCD Library	1114 bytes	1364 bytes
Total Flash Size	2092 bytes	3958 bytes

Table 10.5: Detail of Optimized Flash Usage in Hello Lawrence Application

The results in the table show that with the projected savings from the shared code elimination optimization, the Haskino version of the application would only have a 22% increase in flash storage required over the C version. This is a much preferred result as compared to the original 200% increase detailed in Section 10.1.2.

Given the opportunity for a large percentage reduction in flash usage by eliminating the duplicated code, finding a method to eliminate the inlining of these functions would provide great benefits. One possible method that would provide these benefits is examined in Chapter 11.

10.2 Case Study: Bootstrapping Haskino

A common mechanism used to demonstrate the flexibility of a language system is to build part of that system using the system itself. An example of this is the GHC Haskell compiler, which is written in Haskell, and a new version is built using an older version of the compiler.

To demonstrate Haskino's ability to perform a partial bootstrapping, I have written a version of the Haskino interpreter in Haskino. This interpreter may then be compiled to C, then machine code, and programmed to flash on the Arduino. It interoperates with the Haskino host, allowing programs written in Haskino to be executed on the interpreter.

The Haskino interpreter's structure, in both implementations, consists of of 4 major sections. The first is the reception of the interpreter primitive over the serial port, and verifying the commands integrity with a checksum calculation. Second, the command byte is used to dispatch processing specific to that primitive. Third, the specific primitive processing verifies the arguments to the command, if any, and calls the appropriate Arduino native API. Finally, if the primitive is a procedure, and requires a response to be sent to the host, that response is built and send over the serial port.

The expression evaluation functions in the C version of the Haskino interpreter, which evaluate the `Expr` portions of the Haskino language, are mutually non-tail recursive. This would be difficult to write in the current version of Haskino, without implementing lambdas in the language, or using some other method of generating C functions with recursive calls. Therefore, the version of the interpreter implemented for this bootstrapping case study is one that implements the shallow version of the interpreter, similar to the original Haskino interpreter described in Chapter 3. The major difference between the updated shallow version, and the one described in Chapter 3, is that the updated interpreter uses the current Haskino protocol, which has been updated to handle deeply embedded primitives. The consequence of this is that most of the primitive handlers in the interpreter need to check that the arguments passed to the primitive are simple shallow literals, not more complicated expressions, before the handler operates on them.

10.2.1 Checksum Calculation

The messages sent to and from the interpreter are protected by a checksum value to provide a message integrity measure. The checksum is additive, and covers all of the bytes of the message except for the checksum byte itself and the ending frame byte.

The checksum must be calculated both on reception of a message, to compare to the value received for the checksum and validate the message, and on transmission of reply messages. In the case study implementation, both the reception and transmission of messages was written in a tail recursive style. The current plugin limitation of transforming functions with only zero or one arguments provided some challenges to writing the the checksum calculation code. Two different methods were used to circumvent these challenges in the case study implementation.

For the receive checksum calculation, a `remoteRef` was used as an accumulator, updated for each received, decoded character. This requires creating the `remoteRef` at the top level of the program, and passing it down to the `readFrame` routine, similar to what we did in Section 10.1. The code for the `readFrame` routine is shown below.

```
readFrame :: RemoteRef Word8 -> Arduino [Word8]
readFrame ref = do
    writeRemoteRef ref 0
    readFrame' []
  where
    readFrame' :: [Word8] -> Arduino [Word8]
    readFrame' l = do
        c <- readChar
        if c == hdlcEscape
        then do
            c' <- readChar
            ch <- readRemoteRef ref
            writeRemoteRef ref (ch + c' `xor` hdlcMask)
            readFrame' $ l ++ (c' `xor` hdlcMask : [])
        else do
            if c == hdlcFrameFlag
            then do
                ch' <- readRemoteRef ref
                checkFrame l ch'
            else do
                ch'' <- readRemoteRef ref
                writeRemoteRef ref (ch'' + c)
                readFrame' $ l ++ (c : [])
```

For the transmit checksum calculation, a different approach was taken to work around the one argument limit to a tail recursive function transformation. To calculate the checksum on transmit, the checksum accumulator is added to the front of the list representing the outgoing message. The process of transmitting a byte then removes two bytes from the list, the checksum and the actual byte to transmit, and then adds the update checksum back. The code which implements transmitting the reply message (without the frame flag) is shown below.

```
sendReplyBytes :: [Word8] -> Arduino ()
sendReplyBytes l = sendReplyBytes' $ 0 : l
  where
    check :: [Word8] -> Word8
    check l' = head l' + l' !! 1

    sendReplyBytes' :: [Word8] -> Arduino ()
    sendReplyBytes' l' = do
        sendEncodedByte $ l' !! 1
        if length l' == 2
        then sendEncodedByte $ check l'
        else sendReplyBytes' $ check l' : drop 2 l'
```

10.2.2 Resource Usage Comparison

Table 10.6 shows the differences in RAM and Flash usage between the two versions, simply measured by the output of the Arduino `avr-size` tool which is run following the program linking. For the Haskino version, the static definition of the task control block was removed before compiling, as it includes the stack space for the default task allocated by the runtime, and stack size is not included in the pure C version by the `avr-size` tool. Stack usage was calculated by passing `-fstack-usage` to `gcc` during compilation, and then analyzing the call tree to find the maximum approximate static usage. Both versions use memory buffers to store the incoming and outgoing messages, and those configurable buffer sizes were set to only handle the maximum size message required by the interpreter. The Haskino version uses the dynamic memory management described in Section 6.7, and the minimum number of allocation blocks required for the interpreter was determined through experimentation, and those buffer sizes are reflected in the numbers in the table.

As can be seen from the measurements in the table, the Haskino version uses more Flash and

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino
Flash Size	12428 bytes	23018 bytes
Static RAM Size	534 bytes	741 bytes
Stack RAM Size	51 bytes	50 bytes
Total RAM Size	585 bytes	791 bytes
Uno Flash Usage	37.9%	70.2%
Uno RAM Usage	26.1%	34.2%

Table 10.6: Summary Sizing of Haskino Interpreter Written in C and Haskino

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino
Scheduler	—	84 bytes
Message Buffers	32 bytes	96 bytes
Remainder of Application/Libs	502 bytes	561 bytes
Total Ram Size	534 bytes	741 bytes

Table 10.7: Detail of Static RAM Usage in Haskino Interpreter

RAM than the C version does. The static RAM usage of the two versions is shown in more detail in Table 10.7. The Haskino version requires more buffer space, because of it's list processing, which requires a larger number of intermediate buffers. The Haskino version uses 84 bytes of memory for structures for the scheduler (the task control block and semaphores). These are not strictly required for the interpreter, as there is only one thread of execution, so they could be optimized out. The remaining memory difference between the two versions is the remainder of the applications/libraries/runtime for the Haskino version, which requires 59 more bytes than the C version, which is an 12% increase.

The flash memory usage consists of several different components, and those are detailed in Table 10.8. The Haskino Runtime makes up approximately 15% of the total flash storage requirements of the Haskino version, and this is a flat "tax" that is independent of the application size. Looking at the true application flash storage requirements, we can see that the Haskino version requires approximately 61% more flash storage than the C version does. In Section 10.2.5 of this chapter, we will examine some of the causes of that expansion, and how it might be reduced.

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino
Arduino Libraries	1032 bytes	1032 bytes
Haskino Runtime	—	3602 bytes
Application	11396 bytes	18384 bytes
Total Flash Size	12428 bytes	23018 bytes

Table 10.8: Detail of Flash Usage in Haskino Interpreter

10.2.3 Processing Time Comparison

In addition to comparing the resource utilization for the interpreter case study, the amount of time to process interpreter primitives was compared between the two interpreter versions. The results from measuring two different DSL primitive sequences is shown in Table 10.9. The first primitive sequence consists of sending a single digitalRead command to the interpreter. The second primitive sequence consists of a sequence of 8 digitalRead primitives followed by a queryFirmware primitive to cause the Remote Monad packet to be transmitted. Both the C and Haskino interpreter code was instrumented with time measurement code that starts timing when the first primitive command is received, and stops timing when the last primitive response is sent. The primitive sequences were repeated 1000 times for the measurements, with the total time measured divided by 1000. The measurements were repeated 10 times, and the mean of those 10 times is reported in the table.

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino	Delta
Total Time digitalRead	4.168 ms	4.093 ms	
Communication time	1.042 ms	1.042 ms	
Host time	0.133 ms	0.133 ms	
Processing digitalRead	2.993 ms	2.918 ms	-2.5%
Total Time digitalWrite	8.204 ms	8.222 ms	
Communication time	6.163 ms	6.163 ms	
Host time	0.188 ms	0.188 ms	
Processing digitalWrite	1.853 ms	1.871 ms	1.0%

Table 10.9: Processing Time in Haskino Interpreter

In addition to the total time taken for each primitive sequence, the table shows the amount of communication time on the serial bus for sending the primitive commands and responses for the sequence, at a baud rate of 115200. Also, since the test measures 1000 primitive sequences in a row, for each sequence the Haskell host will take some time to process the response and send out the new primitive sequence. This host time was measured using the debug output on the host, which is timestamped. The mean host processing time is included for each sequence in the table as well. The processing time on the Arduino is the total time for the sequence, minus the communication time and the host processing time.

There is a small difference in processing time between the C and Haskino interpreters in each case. In one case the C interpreter was faster, and in the other the Haskino interpreter is faster. The communication time is a large portion of both examples. At first glance, it appears that the processing time is of the same order for both interpreters, with neither having a clear advantage. The C interpreter may be faster in some cases due to using simple C arrays for buffers, while the Haskino interpreter uses list processing. On the other hand, the inlining of the Haskino generated code will have a performance advantage over the function calls present in the C version.

The timing analysis for this limited case study does not lead to more general conclusions, other than saying both versions are functional within the interpreter requirements, and are able to process the interpreter commands within the same order of magnitude as the communication time.

10.2.4 List Processing Optimization

Much of the computation in the Haskino interpreter involves manipulating the primitive messages sent to and from the host. In the Haskino version, most of these operations are programmed using the `[Word8]` list types that are part of the EDSL.

Upon examining the C code generated from the Haskino compiler, it was found that there were many sections of code which looked like the following:

```
if (((((uint8_t) list8Elem(list8Slice(bind5,1,0),0)) == 2) &&
    ((((uint8_t) list8Elem(list8Slice(bind5,1,0),1)) == 0) &&
    ((((uint8_t) list8Elem(list8Slice(bind5,1,0),3)) == 2) &&
    ((((uint8_t) list8Elem(list8Slice(bind5,1,0),4)) == 0)))))
```

This is the type of code generated from the Haskino source code in the primitive handlers which verify that their arguments are literals. The Haskino code snippet which it is generated from is shown below.

```
processSetPinMode :: [Word8] -> Arduino ()
processSetPinMode m =
    if (head m == exprTypeVal EXPR_WORD8) &&
       (m !! 1 == exprOpVal EXPR_LIT) &&
       (m !! 3 == exprTypeVal EXPR_WORD8) &&
       (m !! 4 == exprOpVal EXPR_LIT)
```

Haskino's expression language was designed to keep the number of basic expression operators to a minimum, to reduce the code required to process those operators in the deeply embedded version of the interpreter. Therefore, Haskell list operations such as `tail`, `take`, and `drop` are all implemented in the Haskino GADT using a `slice` operation, which takes a list and two parameters.

```
SliceList8 :: Expr [Word8] -> Expr Int -> Expr Int -> Expr [Word8]
```

The first parameter specifies the start index of the slice of the list to be taken, and the second parameter specifies the length of slice. If the second parameter is zero, it indicates that all of the list after the start index should be returned. This Haskino `Expr` primitive is translated into `list8Slice` in the Haskino runtime, and the list element operator, `!!`, is translated into `list8Elem` in the runtime.

The repeated occurrences of `list8Elem` with a parameter of `list8Slice` in the generated code are the result of `processSetPinMode` being called with a parameter `m`, which is the result of calling `tail` on the received Haskino primitive. This result of `tail` is the primitive message with the primitive identifier removed. These repeated calls may be optimized by making use of an axiom about the operation of the element and slice operators:

```
forall (list :: [Word8]) (start :: Int) (length :: Int) (index:: Int).
ElemList8 (SliceList8 list start length) index
=
ElemList8 list (start + index)
```

When the Haskino compiler is processing an `ElemList8` operator, it can examine the list parameter, and determine if it is a `SliceList8` operator. If so, the call to `list8Slice` can be eliminated from the generated code. Doing so changes the C code generated for our example to the following.

```
if (((((uint8_t) list8Elem(bind5,(1 + 0))) == 2) &&
    ((((uint8_t) list8Elem(bind5,(1 + 1))) == 0) &&
    ((((uint8_t) list8Elem(bind5,(1 + 3))) == 2) &&
    ((((uint8_t) list8Elem(bind5,(1 + 4))) == 0)))))
```

The generated code could be further simplified by implementing Haskino compiler optimizations for constant folding. However, the `gcc` compiler which is used to compile the C to machine code already has such optimizations built in. Implementing the list splice/element optimization in the Haskino compiler results in the reduction of Flash space used by the Haskino shallow interpreter from 23018 bytes to 21898 bytes, which is a 4.9% reduction.

Two other axioms related to the list processing were subsequently discovered, and are shown below.

```
forall (list :: [Word8]) (start :: Int) (length :: Int) (index:: Int).
LenList8 (SliceList8 list start length)
=
SubInt (LenList8 list) index

forall (list :: [Word8]).
LenList8 (RevList8 list)
=
LenList8 list
```

Optimizations for the Haskino compiler were implemented for these axioms as well. They had only a negligible effect on the case study, but may have a greater effect on other examples where they are used with a greater frequency.

10.2.5 Duplicated Code

After the list processing optimizations have been implemented, there still remains a considerable gap between the flash memory resources used by the Haskino and C versions of the interpreter.

Analysis of the generated code reveales another large contributor to the size of the Haskino version. The structure of the interpreter is such that the receive message code is at the start of the main loop, and is shared by all of the primitive processing. However, the send reply message code is called at the end of each primitive processing branch. Due to the inherent inlining nature of executing a deeply embedded EDSL, this send reply code is repeated in each of the primitive processing branches in the generated code. In the case study interpreter there are 19 copies of the send message code, which is generated from the Haskino source function `sendReply`. Measuring the size of the generated code for this function reveals that it requires 268 bytes of Flash space per instance of the generated code. If this code were moved to a generated C function, 18 copies of it could be eliminated. This would require inserting 19 function calls in their place, and estimating 16 bytes of code per function call, the estimated savings would be 18 * 268 - 19 * 16, or 4520 bytes. Table 10.10 shows the differences between the interpreter version after these savings.

	Shallow Haskino Interpreter in C	Shallow Haskino Interpreter in Haskino
Arduino Libraries	1032 bytes	1032 bytes
Haskino Runtime	—	3602 bytes
Application	11396 bytes	12744 bytes
Total Flash Size	12428 bytes	17378 bytes

Table 10.10: Detail of Optimized Flash Usage in Haskino Interpreter

The results in the table show that with the projected savings from the shared code elimination optimization, the Haskino version of the interpreter would only have a 12% increase in flash storage required over the C version. This is a much preferred result as compared to the original 61% increase detailed in Section 10.2.2.

Given the opportunity for a large percentage reduction in flash usage by eliminating the duplicated code, especially with the limited resources present in an Arduino, finding a method to eliminate the inlining of these functions would provide great benefits. One possible method that would provide these benefits is examined in Chapter 11.

Chapter 11

Sharing in the Generated Code

Section 10.2.5 discussed the issue of the large increase in compiled program size due to the inlining of functions that are called repeatedly. This chapter presents a transformation solution that allows those repeatedly called routines to be compiled into independent C functions to reduce the generated code size. This method has not yet been implemented in the plugin, but the transformation rules and methods for doing so are presented in this chapter.

Eliminating this type of duplicated inlined code has been previously examined, by using a similar method to the one I will describe, and was referred to as λ-sharing in relation to the Nikola EDSL (Mainland & Morrisett, 2010). The Nikola method to eliminate the duplication was similar, however, the Nikola EDSL design is different from Haskino. The EDSL approach used in Nikola is not monadic at the user level (although it does use monads for reification), and it required the user to insert an additional function call into the EDSL source to indicate a lambda that was shared (Haskino's method for identifying the shared functions will be covered in Section 11.3).

11.1 Plugin Transformation for Sharing

The goal of the transformation is to somehow structure the EDSL description of the function such that the compiler is able to know how to generate an independent C function in a type preserving manner. We want to do this without introducing lambdas and higher kinds to the GADT we use to represent the EDSL, as this would greatly complicate the implementation.

To start, we define members of the ArduinoPrimitive GADT, one for each arity of function transformation we wish to support. We could define the application primitives to use currying, however, that would require function types in the GADT.

```
App1Arg :: (ExprB a, ExprB b) => String ->
           ExprArgType a -> ExprRetType b -> ArduinoPrimitive (Expr b)
App2Arg :: (ExprB a, ExprB b, ExprB c) => String ->
           ExprArgType a -> ExprArgType b -> ExprRetType c ->
           ArduinoPrimitive (Expr c)
             ⋮
```

Also, user facing functions are defined to call them. In this case, they are not really user facing, but will only be used by the Haskino plugin in the transformation.

```
app1Arg :: (ExprB a, ExprB b) => String ->
            ExprArgType a -> ExprRetType b -> Arduino (Expr b)
app2Arg :: (ExprB a, ExprB b, ExprB c) => String ->
            ExprArgType a -> ExprArgType b -> ExprRetType c ->
            Arduino (Expr c)
            ⋮
```

These functions make use of two new types which wrap the `Expr` arguments and the `Arduino` monadic return value. Their purpose is to pass type information to the compiler in a easy to use manner. The compiler could determine the type by pattern matching on the expressions and the Arudino primitives that make up the body of the function, however, this would be complicated due to the large number of expression and EDSL primitives that make up Haskino. It would also be fragile, requiring change every time a new primitive is added. Instead, we use the new types in a manner similar to the type parameters of a Core function. We will pattern match on the constructors for these types, which will indicate the `ExprB` type of the expression or the monadic return value. These wrappers are defined as:

```
data ExprArgType a where
    ExprArgTypeB       :: Expr Bool -> ExprArgType Bool
    ExprArgTypeW8      :: Expr Word8 -> ExprArgType Word8
    ⋮
```

```
data ExprRetType a where
    ExprRetTypeB       :: Arduino(Expr Bool) -> ExprRetType Bool
    ExprRetTypeW8      :: Arduino(Expr Word8) -> ExprRetType Word8
    ⋮
```

A new type class and an extension to the ExprB type class are also added to provide polymorphic support for the plugin translation.

```
class ExprB a => ArduinoApp a where
    exprRetType :: Arduino(Expr a) -> ExprRetType a

class ExprB a where
    ⋮
    exprArgType :: Expr a -> ExprArgType a
```

Finally, we add a second new function to the ExprB class, which is used for construction of "remote arguments". This is analogous to the remBind that was discussed in Section 4.1.1.

```
class ExprB a where
    ⋮
    remArg :: Int -> Expr a
    ⋮
```

Now that we have laid the ground work by defining the data structures that will be used to construct the transformed functions by the plugin, let's look at transforming the simple function exampleFunc, which adds two integer arguments.

```
exampleFunc :: Expr Int -> Expr Int -> Arduino(Expr Int)
exampleFunc x y = return $ x + y
```

This function will be transformed to the pair of functions show below.

```
exampleFunc :: Expr Int -> Expr Int -> Arduino(Expr Int)
exampleFunc x y =
    app2Arg "exampleFunc" (exprArgType  x) (exprArgType  y)
                    (exprRetType (exampleFunc_orig (remArg 0) (remArg 1)))

exampleFunc_orig :: Expr Int -> Expr Int -> Arduino(Expr Int)
exampleFunc_orig x y = return $ x + y
```

The original function body is placed in a new function binding. The original binding is re-defined to call the application function of the proper arity, in this case `app2Arg`. The original argument expressions are wrapped by the `exprArgType` constructors. The original body function is called with `Expr` arguments of the `remArg` class function, which have an integer argument that specifies the argument position. When this function is executed during cross-complation, the AST that is built will have these `remArg` expressions substituted at the expression argument sites in the original function, and the compiler will be able to translate these into the proper C function argument names. In other words, this call with evaluate to the following code in our example.

```
return $ (RemArgInt 0) + (RemArgInt 1)
```

Just as we wrapped the arguments with `exprArgType`, the function call of the original function is wrapped with a `exprRetType` to pass the return type information to the compiler.

Replacing the original function with a wrapped call to a helper function provides two advantages over trying to do a transformation at the call site of a transformed function. First, it continues to support the multi-module compilation that the rest of Haskino supports. Second, support of partially applied functions in the Haskino source is more easily supported, as the transformed function will be called when all of the arguments have been provided, and again does not require function types to be added to the GADT.

11.2 Compiler Support

The compiler will be extended to support compiling the new `appXArg` primitives. The string provided as the first parameter to the `appXArg` function will be used as the name of the non-inlined C function generated by the compiler. This will require the function name generated by the plugin to be unique, and may require including a Haskell module identifier in the name as well. (For example `Module.function` may be transformed to `Module_function`).

The compiler will be able to generate the function call at the call cite using this name and it's standard compilation of `Expr` expressions with the arguments. In our example case, if the original call site looked like:

```
exampleFunc (3 * 4) (5 * 6)
```

the generated C code at the call site would be:

```
exampleFunc((3*4), (5*6))
```

The compiler will then need to compile the independent, non-inlined function. It will match on the `ExprArgType` and `ExprRetType` to generate the type signature of the function. As the original function may be called many times, the compiler will need to keep a list of those functions which have been compiled, and only compile the function body the first time one of it's call sites is encountered. It will also need to extend it's `CompileState` data type, so that it may maintain a list of compiled functions which will be added to the output before the body of the main task function.

For our example function, the complier would generate a C function like the following:

```
int exampleFunc(int arg1, int arg2)
    {
    return(arg1 + arg2)
    }
```

11.3 Designating Functions for Sharing

There are several potential methods for determining which functions would need to be independently generated, as opposed to inlined. GHC supports the ability of the user to mark a function with an annotation. One intended use of these annotations is for compiler plugins such as ours. The user could mark those functions as needed to not be inlined with such annotations. Conversely, the default compilation could be flipped to be non-inlined, and the user could mark functions to be inlined.

Finally, the plugin could employ a heuristic, making one or more passes through the code, to determine if a function is called more than a specified number of times (or using another appropriate metric), and generate separate functions for those that satisfy the heuristic.

11.4 Haskino Foreign Function Interface

The same transformation used for un-inlining of code would also prove to be very useful for defining a Foreign Function Interface (FFI) for use with Haskino. This would allow Haskino functions to call C library functions which have been linked with the generated C code.

First, we will define EDSL primitives and a type class which will be used as a placeholder for the external function body in the Haskino source.

```
ArduinoFFIB :: ArduinoPrimitive (Expr Bool)
ArduinoFFIW8 :: ArudinoPrmitive (Expr Word8)
    ⋮

class ExprB a => ArduinoFFI a where
    arduinoFFi :: Arduino(Expr a)
```

Consider the following example in Haskino source, where a user could include the following code to designate that an example C function which takes a `uint8_t` argument and a `uint16_t` argument, and returns a `uint32_t` result, is an external C function.

```
exampleCFunc :: Expr Word8 -> Expr Word16 -> Arduino(Expr Word32)
exampleCFunc = arduinoFFI
```

The plugin could detect the binding to `arduinoFFI`, and transform this into:

```
exampleCFunc :: Expr Word8 -> Expr Word16 -> Arduino(Expr Word32)
exampleCFunc x y =
    app2Arg "exampleCFunc" (exprArgType x) (exprArgType y)
                (exprRetType arduinoFFI)
```

The compiler, when compiling an `appXArg` primitive, would compile the function call at the call site in the same manner as it did in Section 11.2. However, instead of compiling the independent function body, it would pattern match on the `arduinoFFI` primitives, and compile an external function definition as follows.

```
extern uint32_t exampleCFunc(uint8_t arg1, uint8_t arg2);
```

One limitation on the external C functions, would be that those functions having arguments or return values with the equivelent of `[Word8]` types would be required to use the Haskino runtime list functions to compute the list operations.

Chapter 12

Related Work

12.1 Functional Languages and Embedded Systems

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 was Fran, Functional Reactive Animation (Elliott & Hudak, 1997; Elliott et al., 2003). It is used to compose rich, multimedia animations using Haskell. Although this is not directly in the embedded systems space, it does illustrate early techniques for compiling deep embedded DSLs.

A recent example is the Ivory language (Elliott et al., 2015) which 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 only generates C rather than also providing a remote interpreter. Also, it's syntax is typical of a deep EDSL and requires additional keywords and structures above idiomatic Haskell. An additional EDSL built on top of Ivory, called Tower (Hickey et al., 2014), 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 Feldspar project (Axelsson et al., 2010, 2011; Svenningsson & Axelsson, 2013) is a 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 (Persson et al., 2012; Svenningsson & Svensson, 2013; Sculthorpe et al., 2013).

A shallowly embedded DSL for programming the Arduino in the Clean language, called ArDSL has been developed (Koopman & Plasmeijer, 2015). Their work does not make use of the remote monad design pattern, and does not provide a tethered, interpreted mode of operation.

The frp-arduino package (Lindberg, 2015) provides a method of programming the Arduino using Haskell, but using a functional reactive programming paradigm, and once again only compiling to C code.

A second method for programming an Arduino with functional reactive programming, this time using F#, is Juniper (Helbling & Guyer, 2016). Juniper is an extensive compiled language, not a DSL. It does not include interpreted capabilities.

12.2 Blending Shallow and Deep EDSLs

There have been several other efforts to blend shallow and deep EDSL's.

Svenningsson and Axelsson (Svenningsson & Axelsson, 2013) explored combining deep and shallow embedding. They used a deep embedding as a low level language, then extended the deep embedding with a shallow embedding written on top of it. Haskell type classes were used to minimize the effort of adding new features to the language.

Yin-Yang (Jovanovic et al., 2014) provides a framework for DSL embedding in Scala which uses Scala macros to provide the translation from a shallow to deep embedding. Yin-Yang goes beyond the translation by also providing autogeneration of the deep DSL from the shallow DSL. The focus of Yin-Yang is in generalizing the shallow to deep transformations, and does not include recursive transformations.

Scherr and Chiba (Scherr & Chiba, 2014) proposed using load time implicit staging, as opposed to compile time mechanisms, as an alternative to deep embedding. Their prototype in Java allows the user to write in a shallow EDSL, then extracts expression semantics from Java bytecode at load time.

Forge (Sujeeth et al., 2013) is a Scala based meta-EDSL framework which can generate both shallow and deep embeddings from a single EDSL specification. Embeddings generated by Forge

use abstract `Rep` types, analogous to my EDSL's `Expr` types. Their shallow embedding is generated as a pure Scala library, while the deeply embedded version is generated as an EDSL using the Delite (Brown et al., 2011) framework.

Both Yin-Yang and Delite are built on top of Lightweight Modular Staging (Rompf & Odersky, 2010), a general purpose staging framework for developing deep EDSL's based on type directed transformations. Conal Elliott's work in compiling to categories (Elliott, 2017) is another method for developing a programing system that exhibits attributes similar to Haskino, combining ease of use with analysis and optimization. Elliott developed GHC plugins (Elliott, 2015a)(Elliott, 2016) for compiling Haskell to hardware (Elliott, 2015b), using worker-wrapper style transformations equivalent to the `abs` and `rep` transformations used in the Haskino plugin. These plugins were later generalized to enable additional interpretations (Elliott, 2017).

Chapter 13

Conclusion

Programming small, embedded systems with a functional programming language can prove challenging. Their limited memory resources do not lend themselves well to the garbage collection of most functional language systems. Embedded domain specific languages provide a way to bridge that gap.

This was the rationale for developing the Haskino programming system. Starting with an interpreter on the Arduino, and using the Remote Monad design pattern to implement monadic communication from the Haskell host, a method for tethered programming of the Arduino with a shallow EDSL was developed. This was subsequently extended to use a deep EDSL, allowing entire blocks of computation to be executed remotely. To overcome the limitation on program size due to limited Arduino resources and the size of the Haskino interpreter, a complimentary compiler was developed that is able to compile the same deep monadic Haskell code used by the interpreter into C code. The C code only requires a small runtime library, and takes up much less of the limited storage resources than the interpreter, allowing more complicated programs to be developed, and also allowing the programmed Arduino to operate standalone.

As many programs for embedded systems are more efficiently implemented with multiple threads of execution, both the Haskino interpreter and compiler allow development of multi-threaded software. The scheduler for the Haskino interpreter provides cooperative scheduling between tasks, as well as intertask communication. The scheduling of multiple threads and inter-thread communication is implemented to work in the same manner in the Haskino runtime used with the compiler. This allows multi-threaded programs to be tested and debugged using the interpreter, then compiled to an executable binary for stand alone execution and deployment.

The completed interpreter and compiler provided complimentary, but effective ways of using Haskell as a development environment for Arduino software. The interpreter's shallow EDSL hosted in Haskell allows the programmer using the EDSL to write in relatively idiomatic Haskell, and provide a quick turnaround development environment. The compiler's deep EDSL provides better performance and resource utilization by allowing code generation from the DSL's abstract syntax tree, although at the cost of a much more difficult to use syntax. The ideal would be to allow the programmer to write in the shallow EDSL, and have it automatically transformed into the deep EDSL, allowing ease of use as well as the benefits of compilation. This has been achieved with the GHC compiler plugin developed for Haskino. This method and plugin, however, are not limited to Haskino, but may be applied to a wide range of similarly structured monadic EDSLs. With one set of source code, an EDSL user is provided with a quick turnaround, prototyping environment, and a higher performance, generated code system.

13.1 Reflections

Haskino began it's life as a method to explore the use of the Remote Monad in the embedded systems space. For that purpose, it was very useful, however, it did not end there. It slowly grew into a useful system for developing software for the Arduino platform using Haskell. Beyond that, it provided an invaluable platform for exploring transformations of EDSLs from shallow to deep.

Development of the plugin using manipulation of Haskell's intermediate language, Core, proved challenging, but this should not have been surprising. Intermediate languages are not designed for general purpose programming, but are used by a much smaller, select community as a language that is useful to optimize compilation. Therefore, documentation and debug techniques are similarly smaller in scope. As the plugin was developed, experience provided clues to recognizing the root cause of issues. Two common issues involved the compiler issuing an error message indicating that a symbol was not found. The first common cause of this was an error in the type that the plugin code had provided for looking up a Core dictionary. If the type did not have an instance of the type class for which the dictionary was being looked up, the compiler would return a non-existent

mangled dictionary name. The errant type is normally part of the mangled name provided in the error message, so with experience it became easier to find the source of the type error. The second common cause of the missing symbol error was when the plugin transformation had inadvertently eliminated a symbol. In this case, the debugging routines described in Section 9.13 were useful, by allowing the dumping of Core from the input to the plugin pass where the error occurred.

Another common error encountered during development was when the transformation was incomplete, and an `abs` was left in the transformed Core. When attempting to compile the program to C, this would result in an error message when the `abs` was evaluated. Once again, the debugging routines could be used to dump the transformed Core, and the offending `abs` could be found. A useful extension to the plugin would be to add source line annotations to the `abs` function as an additional parameter. Similar annotations to an EDSL using a GHC plugin have been proposed before (Seidel, 2014). This would allow for a more informative error message which would indicate a relatively specific location in the Haskino source where the issue occurred.

The types chosen to be a part of the Haskino `ExprB` type class, and therefore part of Haskino's `Expr` language, and transformable by the plugin, were initially limited to the unsigned integers, signed integers, floats, and booleans that are used by the Arduino API. The Arduino API also includes an enumerated type, PinMode, with three defined constructors, `INPUT`, `OUTPUT`, and `INPUT_PULLUP`. The initial versions of translated Haskino did not include this type in the expression language, but instead left the the values passed to the `setPinMode` primitive as a shallow value, the result of a `fromEnum` call on the PinMode value. This proved sufficient for basic programs, but when implementing the Haskino interpreter in Haskino, it was not acceptable. It was desirable to write the code to handle the `setPinMode` command with a code snippet like the one below.

```
let mode = case m !! 5 of
              0 -> INPUT
              1 -> OUTPUT
              2 -> INPUT_PULLUP
              _ -> INPUT
setPinMode (m !! 2) mode
```

This code failed to translate, with an error caused by an `abs` left in the translated code. The shallow values could only be evaluated and bound to a variable on the host, not on the target as part of the generated deep code. Because of this, `PinMode` was added as a full fledged member of the `ExprB` type class.

13.2 Future Work

The Haskino plugin is currently designed to be used with different monadic EDSLs, by using tables with the plugin to specify the type, primitives, and operations in the EDSL. This could be improved upon by work to further generalize the system, so that it could be used with both monadic and non-monadic EDSLs.

The plugin currently handles translation of monadic code using binds. A natural extension of this would be to develop transformation rules to handle other higher level functions such as mapM. These would then be used to extend the plugin system for working with EDSLs implementing primitives for these functions.

The list processing optimization described in Section 10.2.4 is one example of optimization that was performed on the C code generated by the plugin and compiler. At the current time, the potential optimizations have not been extensively explored, and this is an area for future research.

Finally, effort could be expended to generalize the tools and routines used within the transformation plugin, to provide a more extensive framework or toolkit for writing such plugins.

References

arduino.cc (2017). `https://www.arduino.cc/en/Reference/Board`.

Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., & Vajdax, A. (2010). Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE'10* (pp. 169–178).

Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., & Persson, A. (2011). The design and implementation of feldspar. In *Implementation and Application of Functional Languages* (pp. 121–136). Springer.

Bracker, J. & Gill, A. (2014). Sunroof: A monadic DSL for generating JavaScript. In M. Flatt & H.-F. Guo (Eds.), *Practical Aspects of Declarative Languages*, volume 8324 of *Lecture Notes in Computer Science* (pp. 65–80). Springer International Publishing.

Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M., & Olukotun, K. (2011). A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11 (pp. 89–100). Washington, DC, USA: IEEE Computer Society.

Dawson, J., Grebe, M., & Gill, A. (2017). Composable network stacks and remote monads. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017 (pp. 86–97). New York, NY, USA: ACM.

Elliott, C. (2013). Hackage package `boolean-0.2.3`.

Elliott, C. (2015a). `https://github.com/conal/lambda-ccc`.

Elliott, C. (2015b). `https://github.com/conal/talk-2015-haskell-to-hardware`.

Elliott, C. (2016). `https://github.com/conal/reification-rules`.

Elliott, C. (2017). Compiling to categories. *Proc. ACM Program. Lang.*, 1(ICFP).

Elliott, C., Finne, S., & de Moor, O. (2003). Compiling embedded languages. *Journal of Functional Programming*, 13(2).

Elliott, C. & Hudak, P. (1997). Functional reactive animation. In *International Conference on Functional Programming*.

Elliott, T., Pike, L., Winwood, S., Hickey, P., Bielman, J., Sharp, J., Seidel, E., & Launchbury, J. (2015). Guilt free ivory. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell* (pp. 189–200).: ACM.

Erkok, L. (2014). Hackage package `hArduino-0.9`.

Farmer, A., Sculthorpe, N., & Gill, A. (2015). Reasoning with the HERMIT: tool support for equational reasoning on GHC core programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell* (pp. 23–34).: ACM.

GHC Team (2016). *The Glorious Glasgow Haskell Compilation System User's Guide, Version 8.0.1*. `http://www.haskell.org/ghc`.

Gill, A., Bull, T., Farmer, A., Kimmell, G., & Komp, E. (2013). Types and associated type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. *Higher-Order and Symbolic Computation*, (pp. 1–20).

Gill, A. & Dawson, J. (2016). Hackage package `remote-monad-0.2`.

Gill, A. & Hutton, G. (2009). The worker/wrapper transformation. *Journal of Functional Programming*.

Gill, A. & Scott, R. (2015). `https://github.com/ku-fpg/blank-canvas`.

Gill, A., Sculthorpe, N., Dawson, J., Eskilson, A., Farmer, A., Grebe, M., Rosenbluth, J., Scott, R., & Stanton, J. (2015). The remote monad design pattern. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell* (pp. 59–70).: ACM.

Girard, J.-Y., Taylor, P., & Lafont, Y. (1989). *Proofs and Types*. New York, NY, USA: Cambridge University Press.

Grebe, M. (2017a). `https://github.com/ku-fpg/haskino`.

Grebe, M. (2017b). `https://github.com/ku-fpg/haskino-examples`.

Grebe, M. & Gill, A. (2016). Haskino: A remote monad for programming the arduino. In *Practical Aspects of Declarative Languages* (pp. 153–168).: Springer.

Grebe, M. & Gill, A. (2017). Threading the Arduino with Haskell. In *Post-Proceedings of Trends in Functional Programming*. inpress.

Grebe, M., Young, D., & Gill, A. (2017). Rewriting a shallow dsl using a ghc compiler extension. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017 (pp. 246–258). New York, NY, USA: ACM.

Helbling, C. & Guyer, S. Z. (2016). Juniper: A functional reactive programming language for the arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*, FARM 2016 (pp. 8–16). New York, NY, USA: ACM.

Hickey, P. C., Pike, L., Elliott, T., Bielman, J., & Launchbury, J. (2014). Building embedded systems with embedded dsls. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming* (pp. 3–9).: ACM.

Hoare, C. A. (1972). Proof of correctness of data representations. *Acta Informatica*, 1(4), 271–281.

Jones, S. & Launchbury, J. (1991). Unboxed values as first class citizens in a non-strict functional language. *Conference on Functional Programming ...*.

Jones, S. P., Tolmach, A., & Hoare, T. (2001). Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop 1* (pp. 203–233).

Jovanovic, V., Shaikhha, A., Stucki, S., Nikolaev, V., Koch, C., & Odersky, M. (2014). Yin-yang: Concealing the deep embedding of dsls. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014 (pp. 73–82). New York, NY, USA: ACM.

Koopman, P. & Plasmeijer, R. (2015). A shallow embedded type safe extendable dsl for the arduino. In *Trends in Functional Programming* (pp. 104–123).: Springer.

Lindberg, R. (2015). Hackage package `frp-arduino-0.1.0.3`.

Mainland, G. & Morrisett, G. (2010). Nikola: Embedding compiled gpu functions in haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10 (pp. 67–78). New York, NY, USA: ACM.

Marlow, S., Brandy, L., Coens, J., & Purdy, J. (2014). There is no fork: An abstraction for efficient, concurrent, and concise data access. In *International Conference on Functional Programming* (pp. 325–337).: ACM.

Persson, A., Axelsson, E., & Svenningsson, J. (2012). Generic monadic constructs for embedded languages. In A. Gill & J. Hage (Eds.), *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science* (pp. 85–99). Springer Berlin Heidelberg.

Peyton Jones, S. & Santos, A. L. M. (1998). A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3), 3–47.

Rompf, T. & Odersky, M. (2010). Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on*

Generative Programming and Component Engineering, GPCE '10 (pp. 127–136). New York, NY, USA: ACM.

Scherr, M. & Chiba, S. (2014). Implicit staging of edsl expressions: A bridge between shallow and deep embedding. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586* (pp. 385–410). New York, NY, USA: Springer-Verlag New York, Inc.

Sculthorpe, N., Bracker, J., Giorgidze, G., & Gill, A. (2013). The constrained-monad problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (pp. 287–298).: ACM.

Seidel, E. (2014). (ab)using compiler plugins to improve embedded dsls. `https://galois.com/blog/2014/12/abusing-compiler-plugins-improve-embedded-dsls/`.

Sujeeth, A. K., Gibbons, A., Brown, K. J., Lee, H., Rompf, T., Odersky, M., & Olukotun, K. (2013). Forge: Generating a high performance dsl implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13 (pp. 145–154). New York, NY, USA: ACM.

Sulzmann, M., Chakravarty, M. M. T., Peyton Jones, S., & Donnelly, K. (2007). System F with type equality coercions. In *Types in Language Design and Implementaion* (pp. 53–66).: ACM.

Svenningsson, J. & Axelsson, E. (2013). Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming* (pp. 21–36). Springer.

Svenningsson, J. D. & Svensson, B. J. (2013). Simple and compositional reification of monadic embedded languages. In *Proceedings of the 18th International Conference on Functional Programming* (pp. 299–304).: ACM.

Wadler, P. & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of*

the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89 (pp. 60–76). New York, NY, USA: ACM.

Ömer Sinan Ağacan (2015). Hackage package `CoreDump-0.1.2.0`.