CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Pattern Matching for Non-inductive Types in Code-generating Haskell EDSLs

Master's Thesis in Computer Science and Engineering

Thomas Li

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Master's thesis 2021

Pattern Matching for Non-inductive Types in Code-generating Haskell EDSLs

Thomas Li

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Pattern Matching for Non-inductive Types in Code-generating Haskell EDSLs
Thomas Li

Supervisor: Alejandro Russo, Department of Computer Science and Engineering
Examiner: David Sands, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Pattern Matching for Non-inductive Types in Code-generating Haskell EDSLs
Thomas Li
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Internet of Things (IoT) devices are becoming increasingly common in the modern home, as are IoT-related security vulnerabilities. There are reasons to believe that many of these vulnerabilities were caused by programming errors made possible due to working in a low-level programming language. The embedded domain-specific language (EDSL) Haski provides a way to write low-level applications in the high-level functional programming language Haskell.

An embedded language inherits many features of its host language with little effort, but some have proven to be more difficult to utilize than others. Haskell's built-in pattern matching is one of its killer features, but also one that is hard to properly integrate into an embedded language. We explore how pattern matching can be implemented in deeply-embedded Haskell EDSLs like Haski, and investigate how to work around fundamental difficulties of pattern matching embedding. We first demonstrate the limits of the current pattern matching solution in Haski and then present the process used to develop the resulting pattern matching framework, including issues encountered and how they were solved. We argue that the framework is applicable to EDSLs with a deeply-embedded design in general, and we integrate the framework into an existing Haski compiler.

Keywords: haskell, haski, edsl, pattern matching, embedded domain-specific languages, functional programming.

Acknowledgements

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

I would like to extend my personal thanks to my supervisor, Alejandro, for your guidance, feedback, and encouragement during the course of this thesis.

To my friends, Robert and Therese, for your invaluable kindness and support.

To my family, for your unending warmth and understanding.

Thomas Li, Gothenburg, August 2021

Contents

List of Figures

List of Listings

List of Tables

1

Introduction

Recent research has indicated that a large number of Internet of Things (IoT) devices have severe security vulnerabilities, making them targets for attackers to gain access to private data, or otherwise partake in malicious activities. Studies have found many different ways in which an attacker could exploit security holes, either for personal gain or for causing economic damage or even physical harm to the attacked party [1, 2, 3]. The possible consequences are not just theoretical but have been observed in real-life attacks: In 2016, over 145 000 smart devices were used in a botnet to perform a massive distributed denial-of-service attack [4], and 2014 saw a computer worm that targeted smart devices and exploited a vulnerability to mine cryptocurrency using the infected hardware [5]. As the field of IoT technology is quickly growing in the consumer market, an increasing number of people will be at risk of being exposed to the aforementioned types of attacks. Thus, the need for better techniques to tackle IoT-related security issues will only grow larger.

Valliappan et al. [6] propose two major reasons for the poor state of security in IoT devices: the programming language typically used for development, and the lack of end-to-end guarantees of data-flows in IoT applications. The programming language Haski was presented in 2020 to address these problems. Haski is an *embedded domain specific language* (EDSL) embedded in Haskell, and is designed to provide a more secure way of programming IoT devices.

Haski adopts core ideas of the earlier synchronous programming language Lustre [7], which uses a *data-flow approach* to programming, where programs are modeled as operations on *streams* of data. Lustre's language design guarantees bounded memory usage for its programs and by using the same synchronous data-flow programming model, Haski is granted the same benefit—useful for its target domain of IoT device programming, where there typically are tight constraints on resource usage.

Using a synchronous stream-based programming style inspired by Lustre has been done previously in other DSLs [8, 9, 10, 11]. Haski does a few things to differentiate itself from other Haskell embeddings. One notable aspect is its clear focus on security, as it incorporates techniques for information-flow control (IFC) into the language itself. Another is the compilation process; Haski embeds a version of Lustre in Haskell, and Haski programs are first compiled to a Lustre representation which is then used to generate the C code.

As for the security aspect, Haski provides the following benefits regarding IoT device

programming:

- Having bounded resource usage by design is useful from a security perspective, as it can help avoid certain memory-related bugs that might lead to vulnerabilities. For example, a program with a memory leak may overflow its memory usage while running, which can lead to potentially dangerous undefined behavior.
- By being designed with IFC in mind, Haski provides built-in support for restricting how sensitive data may propagate within an application. By utilizing Haskell's type system, the compiler can assist in detecting potential leaks of sensitive information.
- Allowing the developer to write their program in a Haskell-embedded DSL reduces the risk of making programming mistakes typical of the target language, which can cause security vulnerabilities, while still generating target language code in the end. For example, if targeting C, an EDSL might provide a safe interface for pointer arithmetic or bit-level operations, which can be prone to user-error otherwise. The increase in safety is both due to the more specialized nature of a DSL which can control or disallow the use of unwanted or unsafe operations, as well as properties inherited from the host language, such as Haskell's strong type system which can minimize type-related bugs.

1.1 Aims and report structure

One of the main tools available to a Haskell programmer is the ability to *pattern match* on values, as a way of defining program control flow and for structuring and destructuring data. Unfortunately, it is not an easy task to embed Haskell's pattern matching mechanism into an EDSL while retaining familiar and clear syntax for the user, such as the use of the `case of` keyword. Haski's current implementation of pattern matching is lightweight and intuitive to use, but has narrow restrictions on the kinds of data that can be pattern-matched on, limiting the usability of Haski as a whole.

This report presents a technique for embedding pattern matching in deeply-embedded, code-generating Haskell EDSLs, by allowing the user to define embedded-level translations from the scrutinee type to a "partition" sum type, the latter of which is pattern matched on. The technique works similar to defining *views* [12] on the scrutinee type, but on an embedded level. As the scrutinee of a pattern match does not get a value until the runtime of the generated program, the branches of pattern matches are "faked" using expressions in the embedded language. This essentially amounts to manually defining the branching conditions of a "match" using embedded boolean expressions, which depend on the *runtime* value of the scrutinee. This requires that the scrutinee is represented by an embedded value, which cannot be used for non-embedded computations. Using the technique therefore demands that the aforementioned translations of the scrutinee be defined using primitive operations for the scrutinee type. Because of this, the technique only supports scrutinees of built-in, non-inductively defined types, such as booleans and numeric types.

The technique is implemented as a generic Haskell framework, and the report demonstrates how it can be applied to the Haski EDSL to improve the expressiveness of Haski programs and the clarity of its generated C code, furthering Haski's purpose as a high-level programming language for its low-level target domain. However, while the development of the framework was done with the domain and use of Haski in mind, the framework is not bound to Haski and the core of the report is the description and analysis of the pattern matching embedding technique used. Therefore, we do not focus significantly on the synchronous data-flow programming model of Haski, nor the IFC aspect. Instead, the subject of this thesis is more closely related to the *embedded* aspect of EDSLs and how an EDSL can make use of the tools provided by its host language.

First, we give a brief description of EDSLs, and why pattern matching embedding in Haskell is a difficult problem. We then demonstrate the limitations of Haski's current method of pattern matching in order to provide motivation for the work performed in this thesis. We detail the embedding technique that we have used and how the pattern matching framework is implemented, as well as the challenges observed during development, with explanations for design decisions made. An example scenario is presented along with the implementation of a sample application, to show how our framework can be used. As Haski's existing pattern matching solution was a motivation for the thesis, we compare it to ours and highlight differences and similarities. We also analyze some inherent advantages and fundamental limitations to our technique, and how it may affect the usability of the framework. Finally, as the design of Haskell-based EDSLs is a rich field, we discuss potential alternative embedding techniques, some notable EDSLs in domains close to that of Haski, and a few other well-known embedding-related challenges.

2

Background

This chapter presents the background of the project and brings the work performed in this thesis into context. A brief introduction of DSLs and Haskell-embedded EDSLs is given, explaining the purpose of domain specific languages and the advantages of using Haskell as a host language. This is followed by an short demonstration of the challenges involved in embedding pattern matching into an EDSL, which should give an intuition of the problem. Some code examples are presented to show the current pattern matching support in Haski, how it is implemented, and how its implementation carries limitations in how it can be used.

2.1 Haskell and GHC

The latest specification of the Haskell language was released in 2010 [13] and lacks several language features used in this project, such as generalized algebraic data types (GADTs) [14] and support for various type-level programming techniques. These features are instead made available as compiler-specific language extensions in the cutting edge Haskell compiler GHC (The Glasgow Haskell Compiler) [15], which both the Haski compiler and the technique presented in this thesis are built with. For convenience, when referring to "Haskell", we implicitly mean the GHC implementation of the language, unless otherwise stated.

2.2 DSLs, EDSLs, and Haskell

A domain specific language (DSL) is designed to be used in a specific *target domain*. In the case of Haski, the domain is the programming of IoT devices. Other examples of DSLs are SQL (domain: managing and querying relational databases), and HTML (domain: designing web documents). Compared to general purpose languages (GPLs), DSLs are typically less flexible, providing only the language constructs and primitives needed for working with the domain in question. In return, DSLs can be more concise and easier to use, compared to using a GPL for the same task. This is due to the language being specialized for the relevant tasks in the domain, leading to less redundancy compared to a GPL, which must support a much wider range of use cases.

Haski is an *embedded* DSL, using Haskell as its *host language*. In an EDSL, programs are written in terms of language constructs of its host language. For example, Haski

programs are encoded as normal Haskell expressions, and the compiler for Haski programs is implemented as a Haskell function. Compiling a Haski program is done by applying this function to the Haskell expression representing the Haski program, the same way as any other function application. Similarly, language constructs and primitives of Haski are implemented as normal Haskell functions. In this sense, an EDSL functions like a library in the host language and is implemented as such, exposing an API for performing a specific set of operations.

Haskell aside, implementing a DSL as an embedding brings some conveniences that can apply regardless of the choice of host language. For example, there is no need to implement dedicated lexing and parsing for an embedded language; because the embedded program is just a construct in the host language, parsing the host language source code will, by definition, parse the embedded language program as well! The same concept can apply for other compilation steps too, such as code generation and type checking; because the embedded language is defined *in terms of* the host language, compiling the host program will automatically perform the same compilation steps to the embedded program as well. Things we "get for free" also extend to tooling and other infrastructure, such as a module system, syntax highlighting in text editors, and interaction with other libraries in the host language. Not all compilation steps can always be borrowed from the host language though, depending on the use case and how the embedded language is implemented. For example, Haski programs need to output C code when run, meaning that a custom code generator needs to be written.

By taking advantage of features from the the host language, the embedded language can enjoy some advantages of the high-level language, without having to implement such features from scratch itself. Some languages have specific properties or features that make them especially suitable for the host language role. For example, Lisp's versatile syntax and macro systems makes it easy to expand the language itself to suit the specific problem being solved [16], traits that are shared by many of its dialects.

Haskell in particular has historically been a popular host language choice, and has served as the base for both industry-grade EDSLs, as well as a wealth of research on EDSLs design (see Chapter 6). One powerful feature of Haskell is algebraic data types (ADTs[1]), which provide an expressive way to model languages, and also serve as a natural representation for abstract syntax trees (ASTs) of programs, making them easy to build and modify. Another core strength is its powerful type system, which can easily be used to benefit an embedded language as well. For example, by using polymorphic types and features such as generalized algebraic data types (GADTs), it is simple to represent expressions in a type-safe manner:

```
add :: Exp Int -> Exp Int -> Exp Int
```

Listing 2.1: Example type signature for an addition function in an embedded language.

[1] Although "ADTs" commonly refers to *abstract* data types, our use of the abbreviation should not cause any ambiguity in this report.

The `add` function operates on values in the embedded language `Exp`, but will be type checked by the Haskell compiler, preventing a user from accidental nonsensical operations such as performing addition on boolean values.

Haski makes extensive use of Haskell's type system, both to increase correctness in its own compiler, as well as to improve correctness in the embedded language. Being an EDSL embedded in Haskell and targeting C, it allows users to write high-level programs in Haskell while generating low-level C code. This way, some common bugs can be avoided more easily, primarily those related to memory and unexpected undefined behavior.

2.2.1 Deep embedding and shallow embedding

The technique of using ADTs to model the AST of a program in an embedded language is called a *deep* embedding. In a deep embedding, the ADT is used as an intermediate representation between the user's code and the final target language output. However, it is also possible to use a different embedding technique, called *shallow* embedding. In a shallow embedding, the language primitives and operators exposed to the user are in direct relation to their semantics, and no intermediate representation is used.

The advantage of shallow embeddings is that they are more lightweight due to the lack of an intermediate representation; this can make them simpler to work with, and sometimes easier to extend with new functionality. In a deep embedding, new language constructs need to be introduced as additional data constructors to the intermediate type, which also means that functions operating on the intermediate type must be modified accordingly. However, deep embeddings are used for EDSLs intended to be compiled to a target language, as code generation in a shallow embedding is impractical and inflexible in comparison. Having an intermediate representation is also useful for performing manipulations to the program (such as optimizations) before code generation, as well as for facilitating multiple backends for different target languages.

As an example, Listing 2.2 defines a minimal deeply-embedded language capable of representing simple addition and multiplication. The expression `(3 * (2 + 1))` would be represented as `Mul (Val 3) (Add (Val 2) (Val 1))`.

```
{-# LANGUAGE GADTs #-}
data Exp a where
    Val :: Int -> Exp Int
    Add :: Exp Int -> Exp Int -> Exp Int
    Mul :: Exp Int -> Exp Int -> Exp Int
```

Listing 2.2: A minimal Haskell EDSL. `Exp` is defined using the `GADTs` GHC language extension to include more type information.

In this EDSL, the `add` function from Listing 2.1 could be defined as simply:

```
add :: Exp Int -> Exp Int -> Exp Int
add = Add
```

The two types of embeddings can be mixed; by keeping the deeply-embedded core set of language constructs small, the compiler backend can be kept smaller as well. Shallow embedding techniques can then be used in addition to the deep embedding, typically to provide more flexible ways of building the deep embedding constructs. Providing new functions that translate to existing primitives in the deep embedding can be an effective way of extending a language without needing to modify the compiler backend. This technique can be seen in [17], as well as in the `caseof` function presented in Chapter 3.

Haski uses a mix of deep and shallow embedding techniques, but has a deeply-embedded core. The pattern matching technique presented in this thesis relies on manipulating an intermediate representation as described above and therefore requires a deep embedding to be present; EDSLs using only shallow embeddings were not taken into consideration.

2.3 The problem of embedding pattern matching

EDSLs embedded in high-level languages can target domains where only lower-level languages might be usable; one benefit of using an EDSL is the ability to bring in abstractions and language features otherwise not available in the target domain. In particular, *pattern matching* is a powerful feature in Haskell that can be used both for program flow control as well as deconstructing and inspecting data structures; a feature lacking in a language such as C.

```
fromMaybe :: a -> Maybe a -> a
fromMaybe defaultVal m = case m of
    Just v  -> v
    Nothing -> defaultVal
```

Listing 2.3: Typical use of pattern matching, performing both value deconstruction and program flow control.

Listing 2.3 shows a typical example of pattern matching in Haskell. In the definition of `fromMaybe`, we pattern match on the *scrutinee* `m`, deconstructing it into a *pattern*, which can be one of the possible constructors for the `Maybe a` type: `Just` and `Nothing`. Furthermore, the program flow depends on which pattern was matched on; we either return a default value, or the value which was bound to `v` in the `Just v` pattern. The use of pattern matching in this manner is fundamental to the language, and is natural to Haskell developers. Ideally, this functionality would be available when working with an embedded language as well, seeing as other features such as type checking transfer naturally to an EDSL.

2.3.1 The compilation process

Unfortunately, for an embedded language like Haski which is intended to be source-to-source compiled from Haskell to C, effectively integrating pattern matching into the embedded language poses a significant challenge. This is because Haskell does not provide an easy way of capturing its built-in behavior of patterns and pattern matching, such that:

1. Patterns and pattern matching can be represented in an internal data structure, which a code generator can use to generate code in C.
2. To the user of the embedded language, using pattern matching in the embedded language is as intuitive as using pattern matching in "normal" Haskell programs.

As described earlier, Haski uses a deep embedding, meaning that a Haski program consists of a Haskell data structure, represented by some core type (such as `Exp`). Compiling a Haski program then, consists of applying a "compile-function" (which will be referred to as `compile`) to this data structure, yielding valid C code as a result. This C code would then, in turn, need to be compiled by a C compiler to produce an executable binary.

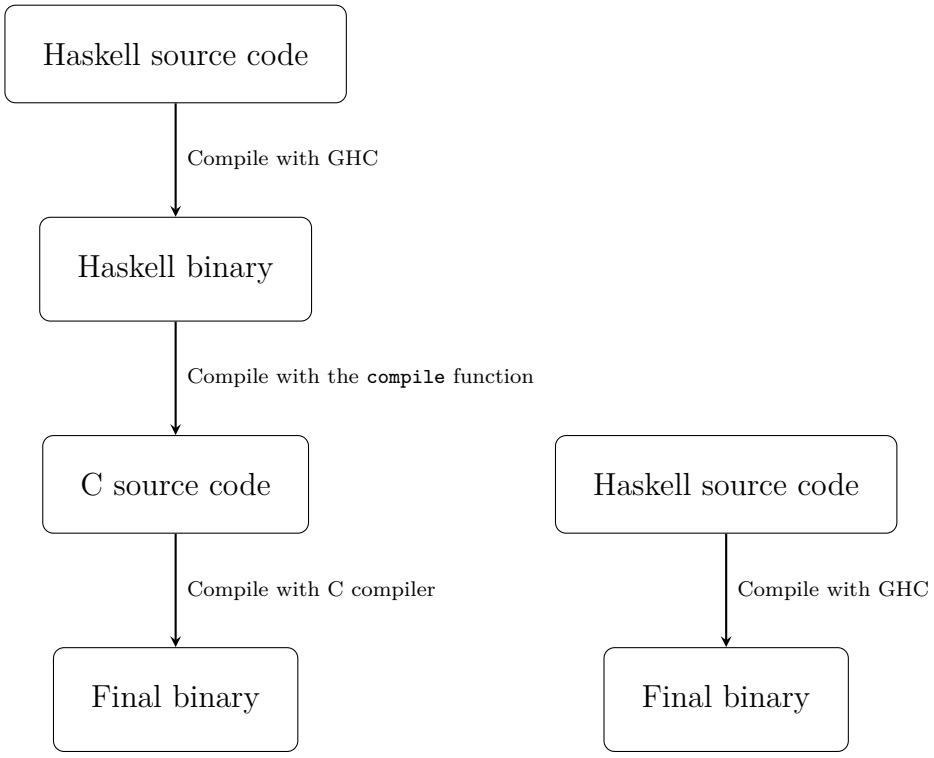

Figure 2.1: A comparison between the compilation process of a typical source-to-source compiler for a Haskell EDSL (left), and a "normal" Haskell program (right).

This compilation process differs from that of a "normal" Haskell program, where the binary produced by the Haskell compiler would be the final product, rather than an intermediate step. Figure 2.1 illustrates the differences.

The `compile` function runs during the Haskell runtime, meaning that we first need to compile the Haskell source code using a Haskell compiler to yield a Haskell executable binary. This binary contains both the compiler (in the form of the `compile` function) as well as the Haski program which is to be compiled (in the form of a Haskell data structure). Running the Haskell binary will output C code corresponding to the semantics of the contained Haski program. Because the output from the Haskell binary is valid C code, it can then be compiled with a C compiler to produce a C binary. This second binary is the final product of the compilation process and is the executable that is intended to be run on the intended target hardware, such as an IoT-device.

An important thing to note here is the difference in purpose between the Haskell code in the two compilation processes. In the "normal" compilation process, the program logic is simply defined directly in the Haskell code, and the compiled Haskell binary will perform the defined actions. In the compilation process for an EDSL like Haski, the Haskell code must additionally also *encode* the program logic in some data structure, such that it can be used by a compiler backend to generate source code in the target language, C in this case.

This embedding step is where pattern matching becomes difficult. While Haskell's ADTs can be used to great effect to embed certain expressions in the style of the example from Listing 2.2, it is difficult to do the same for some of Haskell's built-in mechanics, such as pattern matching.

For an example of the difference, consider the process of embedding addition into the earlier `Exp` type. This can be done simply defining a new operator:

```
(+.) :: Exp Int -> Exp Int -> Exp Int
(+.) e1 e2 = Add e1 e2
```

This custom addition operator operates on embedded values, encoding the addition operation as data, which can be used by a code generator to output corresponding C code. Nonetheless, to the user of the EDSL, it appears as very similar to the default `(+)` operator, and should be intuitive to use. In this case, it would even be possible to completely overload the `(+)` operator, by implementing an instance of the `Num` type class for the `Exp` type.

For pattern matching, it is not as straightforward to implement a corresponding "case of" function. The user would like to be able to use pattern matching in the host language and have its behavior automatically be encoded as a structure in the embedded language. We can compare this to how the `(+.)` operator is applied like a normal Haskell operator but encodes the addition operation with the `Add` constructor. How to implement a solution that accomplishes this for pattern matching, however, is not obvious.

To illustrate the problem of a naive attempt at using embedded pattern matching, we again use the earlier EDSL from Listing 2.2, with some compile-function `compile`:

```
compile :: Show a => Exp a -> String
```

```
fromMaybeExp :: Exp a -> Maybe (Exp a) -> Exp a
fromMaybeExp defaultExp m = case m of
    Just e  -> e
    Nothing -> defaultExp
```

Listing 2.4: An example where pattern matching is not properly embedded into the EDSL. The pattern matching on the `Just` and `Nothing` constructors is performed during the Haskell runtime and is not encoded into an `Exp` value.

Consider the `fromMaybeExp` function in Listing 2.4, which is a corresponding version of the `fromMaybe` function, but (incorrectly) adapted to operate on `Exp` values. The implementation might look reasonable at a first glance. The problem is that the case expression we have used to pattern match on the `Maybe (Exp a)` value is **not** functioning on the embedded level. What we want is for the value deconstruction and flow control introduced by the case expression to carry over to our generated C code. However, when we run `compile (fromMaybeExp (Val 1) x)` for some variable `x`, `x` will have an actual value, either `Just` or `Nothing`. This means that our C code will always contain the body of only *one* of the right-hand sides from the case expression.

What we have is a *staging problem*; the scrutinee is only given a value during the runtime of the generated *target* language program (C), but currently we instead attempt to inspect it during the runtime of our *host* language program (Haskell), specifically when we run `compile`. We need to inspect the scrutinee to perform pattern matching, so if we want to use Haskell's built-in pattern matching, we can only do so when running the Haskell code. However, because the value of the scrutinee is only known at a later stage, when the C code is run, this is not possible. Attempting to solve this mismatch is at the core of a successful technique for pattern matching embedding.

It should be noted that letting the scrutinee in Listing 2.4 have the type `Exp (Maybe a)` instead would raise the question of what a pattern match on such a value would look like; the `Just` or `Nothing` constructor would be hidden "inside" of the embedding. The opaqueness of embedded values is a limiting factor that is discussed in Chapter 5.

2.4 Pattern matching in Haski

Understanding how the Haski compiler handles implementation details specific to the synchronous data-flow programming model is not needed for the scope of this report; as Haski is an embedded language, the problems related to pattern matching embedding discussed in the previous section apply to Haski as well. It suffices to

know that data streams are represented by the core `Stream` type, which for the purpose of pattern matching embedding corresponds to the `Exp` type from earlier.

Haski already supports a limited form of pattern matching, which can be used to define data streams by pattern matching on the values of other streams. Currently, this method is limited to only be usable on values of finitely enumerable types, such as (bounded) integers, booleans, and user-defined enumerable types. This section presents some simple examples to demonstrate why the current method of code generation requires this restriction. For ease of reading, the exact types of functions may be slightly simplified compared their actual implementations, but this does not significantly affect their core behavior.

```
foo :: Stream Bool -> Haski (Stream Int)
foo b = match b f
  where
    f :: Bool -> (Stream Int)
    f x = case x of
        True  -> 99
        False -> 0
```

Listing 2.5: The integer stream `foo` is defined in terms of a boolean stream given by the parameter `b`.

The function `foo` in Listing 2.5 is defined by pattern matching on a stream of boolean values `True` and `False`. The pattern matching is performed using the built-in combinator `match` and the locally defined *matching function* `f`, which uses standard Haskell pattern matching. The type of `match` can intuitively be viewed as

```
match :: Stream a -> (a -> Stream b) -> Haski (Stream b)
```

taking a stream of `a`:s and a function of type `(a -> Stream b)` as its parameters, and outputting a stream of `b`:s.

To compile the pattern matching performed by `match`, Haski generates a switch-statement with cases that cover every possible output value that can be produced by its function argument. This is done by enumerating the entire set of values of its input stream type, and applying the given function to each one to get the set of output values. For example, in `foo`, we generate values `True` and `False`, and apply `f` to get the outputs `99` and `0`. If the value of `foo` was assigned to some variable `a`, this would translate to a switch-case resembling that of Listing 2.6.

In other words, one case is created for each possible value of the input type; this is why the input type must be finitely enumerable. To be specific, a prerequisite for `match` is that the scrutinee type is an instance of the `Enum` and `Bounded` type classes. This limits the possible use cases of `match` as it cannot be used with certain useful types, such as floating-point numbers and non-bounded integers.

However, the situation is not necessarily ideal even for types that do have a finite number of elements. Larger types such as (bounded) integers will cause very big

```
switch (b) {
    case TRUE: a = 99; break;
    case FALSE: a = 0; break;
}
```

Listing 2.6: The switch-statement generated from the code in Listing 2.5.

```
bar :: Stream Int8 -> Haski (Stream Int8)
bar n = match n f
  where
    f :: Int8 -> Stream Int8
    f x = case x > 2 of
        True  -> x - 1
        False -> 0
```

```
switch (n) {
    ... // Cases 5 to 127 omitted
    case 4:    a = 3; break;
    case 3:    a = 2; break;
    case 2:    a = 0; break;
    case 1:    a = 0; break;
    case 0:    a = 0; break;
    case (-1): a = 0; break;
    ... // Cases -128 to -2 omitted
}
```

Listing 2.7: Using `match` on an input stream of signed 8-bit integers, of which there are 256 values.

switch-statements to be generated; Listing 2.7 demonstrates such a situation using the 8-bit signed integer type `Int8`. Although the generated code does work, it is unwieldy and the technique does not scale well; for example, applications of `match` on larger integer types such as `Int64` would yield huge switch-statements and unfeasibly large source code outputs.

3

Methods

This chapter presents aspects needed to be taken into consideration when implementing a pattern matching embedding, and how the implementation process was carried out in this project. First, we describe the ergonomic aspect of an embedded pattern matching implementation. This highlights the balance between ease of implementation and ease of use of the EDSL. We then list the implementation steps taken during the process of developing our pattern matching technique, and the reasoning behind this process. Relevant information regarding the development environment are also detailed, such as the specific compiler and version used.

3.1 Embedding pattern matching

Roughly, embedding pattern matching can be viewed as consisting of two parts: *representation* and *translation*.

As has been mentioned, a typical Haskell EDSL will use some data type that represents the various constructs of our program (such as `Exp` or `Stream`). In order to embed pattern matching in our EDSL, we first need a *representation* of the semantics of pattern matching, containing enough information for our compiler to be able to generate code in our target language. As an example, using our `Exp` type from earlier, we might introduce a constructor representing the behavior of a case expression, resembling the following:

```
data Exp a where
    {- ... -}
    Case :: Exp a -> [(a -> Bool, Exp b)] -> Exp b
```

Indeed, the type of the `Case` constructor appears to capture the core usage of the Haskell `case of` keyword; we inspect some scrutinee of type `a` and return one of several values of type `b`, of which the chosen value depends on the scrutinee (indicated by the `(a -> Bool)` function). Note that the `Exp b` value is a closure that can depend on and use the scrutinee of type `a`, even though it returns a value of type `b`.

However, coming up with a proper representation of pattern matching is only one half of the problem. The second question, which is arguably more difficult, is how

to create an ergonomic user-level interface for our EDSL, and how to *translate* the interface into instances of our pattern matching representation.

There exists a tradeoff between the user-friendliness of the interface of our language and the difficulty of translating it to the internal representation:

(a) At one end of the spectrum, it would be the most convenient for the user to be able to write case expressions like in any normal Haskell program and have them be automatically converted to the internal pattern matching representation, in the form of the `Case` constructor above. This would mean that the user could write code in the EDSL practically identically to any other, non-embedded Haskell program. This adds the least amount of cognitive overhead for the user, who does not have to learn new keywords or functions. However, translating "raw" Haskell into our internal representation is difficult to do from within Haskell.

(b) At the opposite end, we could have the user use the `Case` constructor directly, manually applying the constructor to the desired arguments. This would be trivial to translate to the internal representation, since what the user writes is already in that form. Of course, this is the same thing as not having a user interface at all and entirely misses the point, since the user is no longer using pattern matching at all, and also needs to be familiar with the internal structure of the compiler.

The goal is to find a balance between the two above extremes: a solution that has enough similarity to normal Haskell pattern matching to be intuitive and usable, while still being translatable into a usable representation for the compiler backend. The technique presented in Chapter 4 was developed with this balance in mind.

3.2 Implementation process

While the development of a pattern matching technique was intended for use with Haski specifically, we have seen how the problem of embedding pattern matching into a Haskell EDSL is a more general problem. Therefore, the project was carried out in the following steps:

1. We first defined our own, generic, embedded Haskell expression language, using a deep embedding. This "EDSL" was not specialized for any particular domain, but served only to capture the core properties of a Haskell EDSL with C as its target language. The language contained only basic functionality such as primitives for arithmetic operations and boolean operators. A simple compiler was written to compile programs in this expression language into C code.

2. A representation of pattern matching was then implemented in our language, extending the language with new constructors and primitives as necessary. The compiler was also extended accordingly to validate that our pattern matching representation would be able to function as intended.

3. After a satisfactory pattern matching representation had been implemented in our expression language, the same technique was used to attempt an implementation in the Haski compiler.

There were two main reasons for using this process, rather than immediately developing a new pattern matching technique directly in the Haski compiler:

- The Haski language contains more domain-specific features, not directly related to the goal of our project. For example, Haski's synchronous data-flow aspect requires certain compilation steps not directly related to pattern matching embedding. This increases the complexity of the compiler and introduces overhead and friction when attempting to test pattern matching implementations. By using a simpler language, we could implement different prototypes faster, meaning we could shorten the time between coming up with an idea for a pattern matching technique and the time of having a testable implementation.
- Being able to implement our pattern matching technique in a generic expression language gives an indication of our solution being applicable to deeply-embedded Haskell DSLs in general, without depending on properties specific to Haski.

3.3 Development environment

As the Haski compiler was built with GHC and is dependent on GHC-specific language extensions, the pattern matching technique presented in Chapter 4 also uses GHC. The version of GHC used in examples throughout this report is 8.6.5. Any language extensions used will be included in code examples the first time they appear.

Because of the dependency on GHC, some implementation details in Chapter 4 and the analysis of the results in Chapter 5 should only be considered in the context of GHC, and may not apply if the same technique is used with other Haskell compilers.

4

Implementation

This chapter presents the implementation of our pattern matching technique in detail, as well as a worked example that uses the technique. The example is presented through a hypothetical scenario with a typical smart-device use case, and demonstrates one domain-relevant way of using the technique. An example is also provided to demonstrate that the technique is capable of working with certain types not supported in Haski's current pattern matching solution, such as `Double`. Both the description of the implementation and the worked example uses a generic expression language `E`, rather than the specific embedding used in Haski. The generic expression language serves as a general representation of Haskell-embedded languages, and is used to demonstrate that the pattern matching technique is not dependent on domain-specific details of Haski, but may be applicable to deeply-embedded Haskell DSLs in general. This is shown at the end of the chapter, where by applying the technique mostly unchanged to Haski, we can write and generate programs in a very similar way to that of `E`.

4.1 The generic expression language `E`

This section briefly describes the generic expression language used in the reference implementation of our pattern matching technique, as well as the worked example in Section 4.3. The expression language, `E`, is a straightforward deep embedding defined by the GADT shown in Listing 4.1. For the sake of brevity, not all constructors or types are shown; constructors and types relevant for the pattern matching technique will be introduced as needed in later sections. For the full definition of `E`, see Section A.1.

The type constraints appearing in Listing 4.1 all exist in Haskell's base library, with the exception of `CType`. The `CType` class is a type class for Haskell types that can be represented in C; in other words, it describes how Haskell values and types are translated to their C equivalents. Listing 4.2 shows the definition of the `CType` class. The full list of implemented `CType` instances is given in Section A.2.

```haskell
{-# LANGUAGE GADTs #-}

data E a where
    - Bring a value into the expression language
    EVal :: a -> E a

    - Straightforward operators
    EAdd :: (Num a) =>          E a -> E a -> E a
    EMul :: (Num a) =>          E a -> E a -> E a
    ESub :: (Num a) =>          E a -> E a -> E a
    EGt  :: (Num a, CType a) => E a -> E a -> E Bool
    ELt  :: (Num a, CType a) => E a -> E a -> E Bool
    EEq  :: (Eq a, CType a) =>  E a -> E a -> E Bool
    ENot ::                     E Bool -> E Bool
    EAnd ::                     E Bool -> E Bool -> E Bool
    EOr  ::                     E Bool -> E Bool -> E Bool

    - Bitwise operators and casting
    EShiftL :: (Bits a) => E a -> E Int -> E a
    EShiftR :: (Bits a) => E a -> E Int -> E a
    EBitAnd :: (Bits a) => E a -> E a -> E a
    ECast   :: (CType a, CType b) => E a -> Proxy b -> E b

    {- ... -}
```

Listing 4.1: Excerpt from the definition of `E`, with constructors supporting representations of basic arithmetic, logic operations, and bitwise operations.

```haskell
{-# LANGUAGE AllowAmbiguousTypes #-}
class CType a where
    ctype   :: String       - Name of type in C
    cval    :: a -> String  - Translation function for values
    cformat :: Char         - Used for printf formatting
```

Listing 4.2: Definition of the `CType` class, which describes how to represent types and values from Haskell in C.

The definition of `E` is very similar to the `Exp` type used for demonstration in Chapter 2 and represents typical expressions in the same way:

```
- Representation in E, and corresponding expressions in C:
x = EMul (EVal 4) (EVal 7)                          - 4 * 7
y = EOr (EGt (EVal 2) (EVal 1)) (EVal False)        - 2 > 1 || 0
z = EBitAnd (EShiftL (EVal 0) (EVal 4)) (EVal 1)    - (0 « 4) & 1
```

For convenience, we define user-facing operators as aliases for the constructors. Some of these operators have names similar to their Haskell/C counterparts:

```
(«.) = EShiftL
(>.)  = EGt
valE  = EVal
notE  = ENot
{- etc. -}
```

We also define instances of the `Num` and `Fractional` type classes in order to use literal syntax for numbers, which lets us write "`8.7`" directly instead of "`EVal 8.7`".

4.2 An overview of the technique

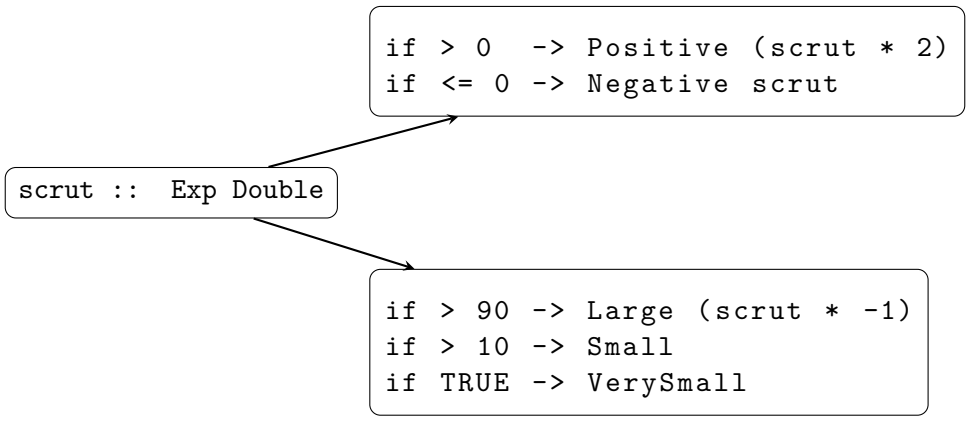

Figure 4.1: Two example translations of an embedded scrutinee with type `Exp Double`. A translation defines a second type whose constructors are pattern matched on during program construction, each with its own associated condition and computation on the scrutinee, here defined in pseudo-code.

The core concept of the technique is to define translations from the type of the scrutinee to a second, user-defined *partition* type. The actual pattern matching that occurs during the Haskell runtime is done only on the second type. This, for example, allows for pattern matching on an interpretation of the scrutinee, even if the scrutinee itself cannot be reliably matched on (such as for floating-point values). To allow for a single combinator to support multiple scrutinee types, each translation on the scrutinee is defined in a separate type class instance, with each instance representing a different translation target.

The setup is similar to unidirectional versions of *views* [12] or the later *pattern synonyms* [18]. The major difference is the need to delay evaluation of the scrutinee until the runtime of the target program, to combat the issue of different compilation stages, described in Section 2.3.

Because of this, the translations on the scrutinee need to be defined in the expression language, so that the translation logic is embedded into the generated code. As the scrutinee cannot actually be pattern matched on, matches are "faked" by separately defining the possible branches for a "match", with each branch defining a branching condition and a return value, both of which can depend on the embedded scrutinee. Essentially, each translation corresponds to a mapping from predicates on the scrutinee to output values, both in the form of embedded expressions. Figure 4.1 illustrates the overall idea.

When the special pattern matching combinator is applied in a program, the user supplies a matching function that performs actual pattern matching on the constructors of some partition type. This function is used in combination with the corresponding translation mapping to construct what effectively amounts to an embedded chain of `if` statements, which is what is embedded into the generated code.

4.3 Usage: A worked example

With our expression language `E` defined, we now present an example to demonstrate how our pattern matching technique can be used in a real-life scenario. The given example is the implementation of a typical smart-device application. Functions and types used in this section represent the user interface of our pattern matching implementation. The details of the implementation, hidden from the user, are described later in Section 4.4.

4.3.1 The scenario

A typical example of a smart device is a sensor that can collect data from its environment, and send the data to some application to process it. Consider a scenario where we are growing plants, and we would like to have the plants watered automatically when needed. We might achieve this by installing a sensor to collect some information about the temperature and soil humidity, and connect the sensor to an application which can then detect whether the plants needs watering.

Sensor data (**16 bits**)																
C	Humidity (%) (**7 bits**)							Temperature (°C) (**8 bits**)								
1	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b

Table 4.1: Bit-level data representation if the control bit C is **set**.

For low-level devices like our sensor, it is common to store data using bit-level encodings, where information is stored within different bits of a single machine word. For example, imagine that the data from our sensor is encoded in a 16-bit integer,

Sensor data (**16 bits**)															
C	Error code (**15 bits**)														
0	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b

Table 4.2: Bit-level data representation if the control bit C is **not set**.

and that it can take one of two forms, depending on whether the most significant bit, the *control bit*, is set:

- If the control bit is **set**, then the data is represented using the pattern shown in Table 4.1. The first 8 bits of the input represent the temperature in degrees Celsius as a signed 8-bit integer, and the following 7 bits represent the humidity percentage of the soil as a 7-bit unsigned integer.
- If the control bit is **not set**, then the sensor is signaling an error, and the remaining 15 bits represent an error code, as shown by Table 4.2.

This specification would be given by the sensor itself and is not in our control as the application developer; this is the data that our program needs to work with.

4.3.2 Writing a program

To write our application, we need to perform some logic based on the input data from the sensor. For example, we might want a function in our expression language that can return `True` if the soil humidity is too low, or if the surrounding temperature is too high.

The type signature for such a function could resemble the following:

```
needsWatering :: E Word16 -> E Bool
```

The problem is that we need to perform a lot of low-level bitwise operations (masking, shifting) to extract the data we want from the encoded input, and these operations could clutter the definition of `needsWatering` which would obscure our application logic. Additionally, the bit-level data representation will not change while the program runs; it will always use one of the two described encodings.

Therefore, a natural next step is to define a Haskell data type to model the two interpretations of the input data, as well as a function to translate the raw sensor data into our type. Listing 4.3 shows a possible definition for such a data type and function.

The `SensorData` type has two constructors, corresponding to the two possible interpretations of our sensor data. The fields of the constructors correspond to the encoded data; the integer types used (`Int8`, `Word8`, `Word16`) are the smallest sizes from the Haskell base library that can fit the data. We can then define a `toSensorData` conversion function to contain the low-level bitwise operations, to keep the rest of the logic of our program clean.

```
type Temp      = E Int8    - Signed 8-bit integer
type Humidity  = E Word8   - Unsigned 8-bit integer
type ErrorCode = E Word16  - Unsigned 16-bit integer

data SensorData
    = Sensor Temp Humidity
    | Error ErrorCode

toSensorData :: E Word16 -> SensorData
{- ... -}
```

Listing 4.3: The two different interpretations of the raw sensor data can be modeled by a Haskell ADT with two constructors, one for each interpretation. The two constructors have different fields, corresponding to the different bit-level patterns depending on the value of the control bit.

```
needsWatering :: E Word16 -> E Bool
needsWatering x = case toSensorData x of
    Sensor temp humidity -> temp >. 30 ||. humidity <. 25
    Error _errorCode     -> valE False  - Placeholder; perform
                                        - error handling here.
```

Listing 4.4: An example of a faulty implementation of our `needsWatering` function; compiling this function into C will not preserve the intended branching logic.

With our new, more convenient form of representing the sensor data, we could attempt to implement our `needsWatering` function as shown in Listing 4.4. We pattern match on the translated sensor data, stating that the plants need watering when the temperature reads above 30 °C or when the soil humidity drops below 25 %. The issue with this implementation is that it runs into the same staging problem described in Section 2.3; when we compile this program, only one of the two branches of our program will be used to generate the final C code, and we lose the intended branching logic of our pattern match.

```
{-# LANGUAGE LambdaCase #-}

needsWatering :: E Word16 -> Estate (E Bool)
needsWatering x = caseof x $ \case
    Sensor temp humidity -> pure $ temp >. 30 ||. humidity <. 25
    Error _errorCode     -> pure $ valE False
```

Listing 4.5: The `needsWatering` function from earlier, rewritten using our presented pattern matching technique.

Here is where we can apply the presented pattern matching technique, rewriting our program to the form shown in Listing 4.5. We note that we apply function

`caseof` in place of `toSensorData` and that we no longer use the built-in Haskell `case of` keywords, instead providing a function defined using a lambda-case. The `\case` notation is syntactic sugar for `\x -> case x of`[1]. The return type is also slightly different; we return a monadic computation `Estate (E Bool)` instead of just a value of type `E Bool`. The `Estate` monad is simply a specialization of the `State` monad and is used when compiling `E` programs to keep track of compilation state, which is used to generate unique identifiers as they become needed.

The key to this pattern matching technique is the `caseof` combinator, whose (simplified) type signature looks as follows:

```
caseof :: Partition a p => E a -> (p -> Estate (E b)) -> Estate (E b)
```

The `caseof` function takes a value of type `E a` as its first argument; this is the scrutinee `x` from the Listing 4.5 example. The second argument is the function that performs pattern matching; in Listing 4.5 this is the function defined by the lambda-case. Note, however, that this function takes a value of type `p` as its input, rather than the scrutinee type `E a`. When `caseof` is applied, it will first transform the `E a` value into a `p` value; this transformation is what is indicated by the `Partition a p` constraint and its purpose corresponds to that of the `toSensorData` function.

For now, we need not pay too much attention to the `Estate` usage in the type signature. As mentioned earlier, `Estate` is used when values need to make use of unique identifiers, but is otherwise not of particular importance for understanding the `caseof` function and `Partition` type class. This applies to later occurrences of `Estate` as well.

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Partition a p where
    partition :: [E a -> (E Bool, Estate p)]
```

Listing 4.6: The definition of the `Partition` type class. An instance of `Partition a p` defines how to convert values of type `a` to values of type `p`.

Listing 4.6 shows the definition of the `Partition` type class. It has one class method, `partition`, which serves two roles for any `Partition a p` instance:

- It defines how to construct values of type `p`, which can depend on some input value of type `a`; this corresponds to the `Estate p` half of the returned tuple.
- It defines a *predicate* on the input value of type `a` for each constructed value of type `p`, indicated by the `E Bool` half of the returned tuple. This captures the branching logic of a case expression; in the generated C code, a branch is chosen and its associated value of type `p` is made available depending on whether the predicate for that branch holds.

[1] Requires the `LambdaCase` language extension.

The idea is that `partition` returns a function for each constructor of the type `p`; the function can use its input value of type `a` to determine *how* to construct a value of type `p`, as well as the condition for *when*. Listing 4.7 shows an example implementation of `partition` for our watering application.

```
{-# LANGUAGE BinaryLiterals        #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NumericUnderscores    #-}

instance Partition Word16 SensorData where
    partition =
        [ \v -> ( testBitE 15 v
                - If the control bit is set,
                - construct a Sensor value!
                , _Sensor (castE (v &. tempMask))
                          (castE (v &. humidityMask ».  8))
                )
        , \v -> ( notE (testBitE 15 v)
                - If the control bit is not set;
                - construct an Error value!
                , _Error (v &. errorMask)
                )
        ]
      where
        - The BinaryLiterals and NumericUnderscores language
        - extensions enable nicer literals for binary numbers.
        tempMask, humidityMask, errorMask :: E Word16
        tempMask     = 0b0000_0000_1111_1111
        humidityMask = 0b0111_1111_0000_0000
        errorMask    = 0b0111_1111_1111_1111
```

Listing 4.7: An example implementation of an instance for the `Partition` class. Note how the types in the instance head correspond to the kind of conversion we want to perform.

Since the `SensorData` type has two constructors, we return two functions, one for each branch. In each of the corresponding `E Bool` fields in our pairs, we implement the check for the control bit in our sensor data. We embed this check using the `testBitE` function, which returns true if the bit at the given index is set. In the second halves of our tuples, we construct values of `p` by applying constructors `Sensor` and `Error` accordingly, expressing the bit-level operations necessary to extract the desired values. Note that we actually use two lookalike-functions `_Sensor` and `_Error`, rather than the actual constructors. These functions can be generated at compile time with Template Haskell by applying the function `mkConstructors` to the `SensorData` type:

```haskell
{-# LANGUAGE TemplateHaskell #-}

data SensorData = Sensor Temp Humidity | Error ErrorCode
$(mkConstructors ''SensorData)
```

The reason for using helper functions in this manner will be covered in Section 4.4, along with other implementation details.

```c
/* ... */
uint16_t _scrut1;
int8_t  _scrut1_1_Temp0;
uint8_t _scrut1_2_Humidity1;

bool v0(uint16_t arg) {
    _scrut1 = arg;
    _scrut1_1_Temp0 = (int8_t) (_scrut1 & 255);
    _scrut1_2_Humidity1 = (uint8_t) ((_scrut1 & 32512) >> 8);

    if (((1 << 15 & _scrut1) != 0)) {
        return _scrut1_1_Temp0 > 30 || _scrut1_2_Humidity1 < 25;
    } else if (!((1 << 15 & _scrut1) != 0)) {
        return false;
    }
}
```

Listing 4.8: The generated C code representing our `needsWatering` program from Listing 4.5. The formatting of the output has been edited slightly for clarity. One might worry that the generated if-statements do not exhaustively cover the possible values of the scrutinee `arg`. This is a valid concern that is discussed further in Section 5.2.3.

With this implementation of `partition`, we can compile the `needsWatering` program from Listing 4.5 to output the corresponding C code output, shown in Listing 4.8. The compiler generates appropriate variable names (`_scrut1_Temp0` and `_scrut1_Humidity1`) to represent each field of our constructor `Sensor` and assigns them values corresponding to our implementation of `partition` for the `SensorData` type.

4.4 Reference implementation

The `caseof` function used in the current form of our pattern matching framework, shown in Section 4.3.2, is a modified version of an earlier implementation attempt. This section describes the design of our pattern matching embedding, and how the implementation of `caseof` was iterated upon in order to solve issues that emerged during development.

4.4.1 The caseof function

```
1 data E a where
2     {- ... -}
3     ECase :: (Partition a p, CType a, CType b)
4         => Scrut a
5         -> [Match p b]
6         -> E b
7 data Scrut a = Scrut (E a) String
8 data Match p b = CType b => Match (E Bool) (E b)
```

Listing 4.9: The definition of the `ECase` data constructor.

In order to embed pattern matching into our expression language and avoid the previously demonstrated staging problem, we need a *representation* of pattern matching that contains enough information for the compiler to be able to output C code corresponding to *all* branches of a pattern match. As such, we extend our `E` data type with the constructor `ECase`, shown in Listing 4.9.

(3–6) The `ECase` constructor represents a pattern match construction, similar to a Haskell case expression. It has two fields: the scrutinee of the pattern match, and a list of *matches*. The `Partition` constraint serves the same purpose as earlier. The `CType` type class has instances for Haskell types that have a corresponding representation in C, including `Int8`, `Bool`, `Double` etc. The `CType` constraint ensures that we only construct values for which we can generate valid C code.

(7) The `Scrut a` type is a simple wrapper for a value of type `E a`, but includes an identifier that is used during code generation.

(8) The `Match` type represents one branch in a case expression. While the `ECase` constructor imposes a `Partition` constraint on the first type parameter, this is only for aiding compilation and increased type safety; values of type `Match p b` do not actually hold values of type `p`. Rather, the fields of its constructor are the same as the tuples returned by `partition`.

The purpose of the `caseof` function is to construct `ECase` values, along with the necessary `Match` values to represent the user-provided matching function (defined with lambda-case in earlier examples). Listing 4.10 shows the definition of one of the earlier versions of `caseof`. The functions `caseof` and `partition` are connected by the `Partition` constraint in the type signature of `caseof`, making it polymorphic over all type combinations that implement a `Partition` instance.

We will refer to the type indicated by the type variable `p` in the type signature as the *partition type*, which is the type that the matching function can deconstruct and pattern match on. How `caseof` works is by mapping the list of functions given by `partition` to a *dummy* scrutinee, resulting in a list of fully applied constructors of the partition type. These are then used to build the `Match` values and the `ECase` construct. The result is a proper expression in the embedded language, but where

```
1  {-# LANGUAGE ScopedTypeVariables #-}
2  {-# LANGUAGE TypeApplications    #-}
3  caseof :: forall p a b . (Partition a p, CType a, CType b)
4      => E a
5      -> (p -> E b)
6      -> E b
7  caseof s f =
8      let varName = "arg"
9          branches = map ($ ESym varName) (partition @a @p)
10     in ECase
11         (Scrut s varName)
12         (map (\(pred, p) -> Match pred (f p)) branches)
13
14 data E a where
15     {- ... -}
16     ESym :: ScrutId -> E a
17 type ScrutId = String
```

Listing 4.10: An early version of the `caseof` function, and the type of the `ESym` "placeholder" constructor.

several of the sub-expressions that originally referred to the pattern match scrutinee have been replaced by placeholder `ESym` constructors (`ESym` for "symbolic variable"). The `ESym` constructor holds a string as a unique identifier, but otherwise contains no information and so can replace an expression of any type. However, since the `ECase` constructor also stores the *actual* scrutinee (the first argument given to `caseof`), the *compiler* can then replace the placeholder expressions with the actual scrutinee before generating the output C code.

(1–2) GHC's `TypeApplications` language extension allows us to specify explicit types for polymorphic functions by prefixing them with the `@`-symbol. The `ScopedTypeVariables` extension allows us to bring type variables from the function type signature into the scope of the body of the function, so that they can be used with `TypeApplications`.

(3–5) The types of `caseof`'s parameters are constrained by its `Partition a p` constraint. Specfically, the scrutinee of type `a` (line 4) must be partitionable into a value of type `p` (line 5) which the matching function can be applied on.

(6) Unlike examples shown earlier, this version of `caseof` does not generate identifiers when constructing `ECase` values, and therefore does not need to be a stateful computation (does not require the `Estate` monad).

(8) We define a name to identify the connection between a symbolic variable and its related scrutinee which it refers to.

(9) The application of `partition` returns a list of functions that can each be applied to the symbolic variable `ESym`. The result is bound to `branches` and

has the type `[(E Bool, p)]`.

(10–12) We apply the matching function `f` to the partition type values `p` in each branch, giving us a list of `Match`es, each representing the body of a case expression branch (the expression to the right of the `->` arrow).

Recall that the problem of applying pattern matching in a deeply-embedded DSL is that we forcibly match one specific pattern during the compilation of our embedded program, which occurs during the runtime of the Haskell host program. By first building the expression language AST using symbolic variables, and only in a later compilation phase substituting in the actual expressions, we can include *all* defined patterns for a scrutinee into our embedded program. Since we define the conditions for "choosing" a pattern as a boolean value in our expression language when implementing `partition`, these conditions can also be included in the embedded program and compiled to the target language; this is what allows us to delay the logic of matching a pattern until the runtime of the embedded program, rather than the runtime of the host program.

It can be observed from the definitions of `caseof` and `partition` that values of the partition type `p` never appear as constructed values in the AST of the embedded language. Rather, they are used only as intermediate values for the matching function to operate on, to create the `Match` values in the `ECase` construct.

```
ex :: E Int -> E Int
ex v = caseof v $ \case
    T2   -> 98
    T1 x -> caseof (x + 1000) $ \case
        T2   -> 99
        T1 y -> y + 2

data T = T1 (E Int) | T2
instance Partition Int T where
    partition = [\v -> (v >=. 0, T1 (v * 2), \v -> (v <. 0, T2)]
```

```
int v1(int arg) {  // Inner pattern match
    if      (arg >= 0) { return arg*2 + 2; }
    else if (arg <  0) { return 99; } }
int v0(int arg) {  // Outer pattern match
    if      (arg >= 0) { return v1(arg*2 + 1000); }
    else if (arg <  0) { return 98; } }
```

Listing 4.11: Compiling `caseof` applications yields one function for each instance of `caseof`. The choice of which pattern to match on is determined by `if`-statements.

At a first glance, this implementation appears to work quite well. Listing 4.11 shows a simple example that compiles sensibly, even with nested matches. A new C function is generated for each match, so the reuse of the "arg" variable name does not cause an issue.

```
ex :: E Int -> E Int
ex v = caseof v $ \case
    T2   -> 98
    T1 x -> caseof (x + 1000) $ \case
    - ^ 'x' was bound in this outer pattern match...
        T2   -> 99
        T1 _ -> x + 2
        -       ^ ...and is still in scope here!
```

Listing 4.12: In the inner pattern match, by referring to a variable that was bound elsewhere, we can introduce incorrect behavior in the generated code. This is because the functions in the generated code only have a single variable (`arg`) in scope, while the scoping is different in the Haskell source code.

However, all is not well; what happens if our inner pattern match refers to a variable that was bound in the outer pattern match? The function in Listing 4.12 compiles to the same C output as the function in Listing 4.11, but this is no longer the correct behavior. In Listing 4.12, because we refer to the `x` bound in the outer pattern match, we expect our inner pattern matching function to increment the *original* input by 2, not the value that has already been increased by 1000. This indicates a clear mismatch between what the user might expect from our expression language, and the actual code emitted by our compiler.

4.4.2 Unique identifiers for scrutinees

```
1  caseof :: forall p a b . (Partition p a, CType a, CType b)
2      => E a
3      -> (p -> Estate (E b))
4      -> Estate (E b)
5  caseof s f = do
6      scrutVar <- freshId
7      let branches = map ($ ESym scrutVar) $ partition @p @a
8      matches <- mapM (\(cond, p) -> Match @p cond <$> f p) branches
9      pure $ ECase (Scrut s scrutVar) matches

10 type Estate a = State Int a  - Alias for specialized State monad
11 freshId :: Estate String     - Return a unique string
12 freshId = ...
```

Listing 4.13: The new version of `caseof` generates a unique variable name each time it is applied, instead of reusing the same identifier `"arg"`.

Thankfully, this problem can be remedied by generating a unique identifier for each application of `caseof` in the program. The compiler can then declare this identifier as a global variable in C, bringing it into scope for all functions. Listing 4.13 shows

the modified `caseof`, and why it must now be run as a stateful computation, as it needs to keep track of an internal counter for generating variable names.

(6–9) Instead of using the same name for all scrutinees in the generated code, we instead generate a unique identifier, `scrutVar`.

(4) We use the State monad to ensure that the identifiers are unique, meaning that `caseof` now returns in a monadic context. The `Estate` name is a type alias for the `State` type carrying an integer in its state.

(3) Because `caseof` now returns in a monadic context, the matching function must also return a monadic computation in order allow convenient syntax for nested applications of `caseof`.

```
ex :: E Int -> Estate (E Int)
ex v = caseof v $ \case
    T2   -> pure 98
    T1 x -> caseof (x + 1000) $ \case
        T2   -> pure 99
        T1 _ -> pure (x + 2)
data T = T1 (E Int) | T2
instance Partition Int T where
    partition = [\v -> (v >=. 0, T1 (v * 2)), \v -> (v <. 0, T2)]
```

```
int _scrut1;
int _scrut0;
int v1(int arg) {
    _scrut1 = arg;
    if      (_scrut1 >= 0) { return _scrut0*2 + 2; }
    else if (_scrut1 <  0) { return 99; } }
int v0(int arg) {
    _scrut0 = arg;
    if      (_scrut0 >= 0) { return v1(_scrut0*2 + 1000); }
    else if (_scrut0 <  0) { return 98; } }
```

Listing 4.14: By generating a unique variable name with `freshId` for each application of `caseof`, we can resolve the scoping described earlier.

Listing 4.14 demonstrates how this change solves the scoping issue; since the scrutinee of a pattern match will now be available in the global scope, nested applications of `caseof` will work as expected. It would also have been possible to modify the inner generated function to take more parameters, so that otherwise out-of-scope variables could be passed in as arguments when called. The decision to use global variables instead was done mostly for the sake of convenience and ease of implementation, and it likely is not an optimal solution.

Unfortunately, there still exists a different problem with this implementation. Given the same definition of `T`, consider the example in Listing 4.15, which refers to the

```
ex :: E Int -> Estate (E Int)
ex v = caseof v $ \case
    T2   -> pure 0
    T1 x -> pure (x * x + x)
```

```
/* ... */
  if (_scrut0 >= 0)
    { return (_scrut0*2) * (_scrut0*2) + (_scrut0*2); }
  else if (_scrut0 < 0)
    { return 0; }
/* ... */
```

Listing 4.15: The value bound to `x` is given by the `*2` operation defined in `partition`. Using `x` multiple times in the `caseof` body causes this operation to be included for each occurrence in the resulting C code.

variable `x` multiple times in the Haskell code. We can see that this is reflected in the C code, but also that the `*2` operation is replicated for each occurrence. This is problematic; the `*2` operation was part of the implementation of `partition`, and is therefore inserted for every reference to the field of `T1`. What we would like is for the computation defined in `partition` to be assigned to variables representing the fields of constructor, which could be reused in the pattern matching function body.

4.4.3 Naming constructor fields

```
data E a where
    {- ... -}
    EField :: String -> E a -> E a
```

Listing 4.16: The `EField` constructor wraps another `E` expression and tags it with a unique identifier.

To accomplish this, we need to somehow determine which expressions in the generated AST of the program correspond to the fields of constructors, "field expressions". A simple way of doing this is to extend the `E` type with a new constructor, with the sole purpose to act as a marker of field expressions. Listing 4.16 shows the `EField` constructor which wraps another `E` value, along with an identifier which is used to derive the variable name in the generated C code. The identifier should be unique to each field expression so that there are no name collisions in the generated code. The actual implementation of `EField` uses a slightly different type for the identifier, but a type like `String` is sufficient.

When an `EField` value is first encountered during compilation, the compiler can know to assign it to the variable given by the identifier. As further field tags with the same identifier are encountered, the wrapped expression can be discarded and the previously defined variable can be used instead.

There is a decision to be made for where in the program construction the `EField` constructor should be inserted. Because there is no obvious indication of whether an expression in the AST is a field expression or not, we can choose between two locations, where we *know* that an expression is a field expression:

- When we implement `partition` for some type, we manually declare field expressions for the constructors of that type. Here, we could simply wrap the constructed field expression with `EField` to have all subsequent values constructed with `partition` be tagged as field expressions.
- When actually deconstructing a value using `caseof`, we know that the bound variables in the pattern correspond to field expressions. This means that we could wrap these variables with `EField` before using them in the body of the branches of the `caseof`.

Both of these options would incur some additional boilerplate code or new syntax, either when implementing `partition` or when applying `caseof`. We chose the first option, the main reason being the follwing: Implementing `partition` instances is already the main source of boilerplate in our pattern matching framework, so requiring a little bit of extra overhead here should be less detrimental for the user experience. This way, our pattern matching trades off a bit of convenience at the "set-up" location (`partition`) for improved usability at the usage location (`caseof`). Furthermore, the boilerplate of defining `partition` only needs to be written once per type to pattern match on, while `caseof` can be used multiple times for that type afterwards.

```
ex :: E Int -> Estate (E Int)
ex v = caseof v $ \case
    T2   -> pure 0
    T1 x -> pure (x * x + x)

data T = T1 (E Int) | T2
instance Partition Int T where
    partition = [ \v -> (v >=. 0, T1 (EField "field1" (v * 2)))
                , \v -> (v <. 0, T2) ]
```

```
int _scrut1, _scrut1_1_field1;
int v0(int arg) {
    _scrut1 = arg;
    _scrut1_1_field1 = _scrut1 * 2;
    if (_scrut1 >= 0) { return _scrut1_1_field1
                             * _scrut1_1_field1
                             + _scrut1_1_field1; }
    else if (_scrut1 < 0) { return 0; } }
```

Listing 4.17: By wrapping field expressions in the implementation of partition, we can save the computed field expression to a variable in the body of generated function `v0`.

Listing 4.17 shows how the example from Listing 4.14 would be defined using this method, along with the generated code. By wrapping the field expression of `T1` with the `EField` constructor and an identifier `"field1"`, the compiler assigns the field expression to a unique variable `_scrut1_1_field1` in the generated code. This variable is then reused in the rest of the function, avoiding the problem of repeatedly computing the value of the field expression—a potentially significant impact on performance if the computation is expensive.

This technique is called *common subexpression elimination* (CSE) and is a commonly used optimization in compilers. Given the maturity of many C compilers, it is possible that they can perform CSE equally well or better compared to the implementation in our EDSL complier, rendering the optimization in our compiler redundant. However, it is also possible that such C compiler optimizations could be less reliable, depending on the exact trigger conditions of the optimizations. For example, it might be less clear to the C compiler whether a given expression can perform side effects or not, leading to more conservative optimization. As developers of the EDSL, however, we can know precisely which operations generate pure code, and which do not, which could allow optimizations to be more precise and aggressive. As the performance aspect was not a main focus of the thesis, we did not benchmark or otherwise compare the performance between using our CSE implementation and leaving it to a C compiler.

While our technique produces satisfactory generated code, it also worsens the boilerplate situation for the user, as they must now manually apply `EField` and supply a identifier when implementing `partition`. This not only introduces noise which decreases readability, but is also potentially error-prone. The user has to manually ensure that the identifiers supplied to `EField` are unique for that constructor, or there will be name collisions during compilation. While it would be possible to have the compiler keep track of such collisions and resolve them, it would be better if the field expression identifiers were generated automatically, like other identifiers such as `_scrut1` and `v0`.

We use smart constructors for this purpose: For each constructor of a type, we create a corresponding function with the same number of parameters, that additionally wraps its arguments with `EField` and a generated identifier. For example, for the constructor `T1 :: E Int -> T`, we create a smart constructor `_T1` similar to the following:

```
_T1 e = do s <- freshId
           return $ T1 (EField s e)
```

The metaprogramming capabilities provided by Template Haskell allows us to generate code during the compilation of the Haskell source code. We use Template Haskell to implement the `mkConstructors` compile-time function, which automatically derives the appropriate smart constructors for types. Listing 4.18 shows the code generated by `mkConstructors` when applied to the type `T`. The smart constructor `_T1` will, when applied to the appropriate arguments, return a *computation* (as indicated by the `Estate T` return type) which will produce a value of type `T`

when run. The `newFieldTag` function has the type `Maybe String -> Estate (E a -> E a)`; given an optional name, it will generate a unique identifier and return a function which wraps an expression with the `EField` constructor and the generated identifier. The name `"E_Int0"` is derived from the definition of `T`; the type of the first field is `E Int`, and the zero indicates the index of the field, since constructors can have multiple fields of the same type.

```
{-# LANGUAGE TemplateHaskell #-}
data T = T1 (E Int) | T2
$(mkConstructors ''T)
```

```
_T1 :: E Int -> Estate T
_T1 e0 = do tagger0 <- newFieldTag (Just "E_Int0")
            return (T1 (tagger0 e0))
_T2 :: Estate T
_T2 = do return T2
```

Listing 4.18: The `mkConstructors` function uses Template Haskell to automatically generate smart constructors for wrapping field expressions with the `EField` constructor.

```
partition :: [E a -> (E Bool, Estate p)]
```

```
caseof :: forall p a b . (Partition a p, CType a, CType b)
    => E a -> (p -> Estate (E b)) -> Estate (E b)
caseof s f = do
    scrutVar <- newScrutId
    let branches = map ($ ESym scrutVar) (partition @a @p)
    taggedBranches <- mapM computeTag branches
    matches <- mapM mkMatch taggedBranches
    pure $ ECase (Scrut s scrutCount) matches
  where
    computeTag :: (E Bool, Estate p) -> Estate (E Bool, p)
    computeTag (cond, p) = (\x -> (cond, x)) <$> p

    mkMatch :: (E Bool, p) -> Estate (Match p b)
    mkMatch (cond, p) = Match @p @b cond <$> f p
```

Listing 4.19: The final versions of the `partition` and `caseof` functions.

The smart constructors generated by `mkConstructors` return a stateful computation rather than a pure value. Therefore, in order to to make use of them, the type of `partition` needs to be updated accordingly, as does the definition of `caseof`. Listing 4.19 show the final versions of these functions.

```
ex :: E Int -> Estate (E Int)
ex v = caseof v $ \case
    T2   -> pure 0
    T1 x -> pure (x * x + x)

data T = T1 (E Int) | T2
$(mkConstructors ''T)
instance Partition Int T where
    partition = [ \v -> (v >=. 0, _T1 (v * 2))
                , \v -> (v <. 0, _T2) ]
```

```
int _scrut1;
int _scrut1_1_E_Int0;
int v0(int arg) {
    _scrut1 = arg;
    _scrut1_1_E_Int0 = _scrut1 * 2;
    if (_scrut1 >= 0) { return _scrut1_1_E_Int0
                             * _scrut1_1_E_Int0
                             + _scrut1_1_E_Int0; }
    else if (_scrut1 < 0) { return 0; } }
```

Listing 4.20: By using smart constructors in the implementation of `partition`, we get reusable variables in the generated C functions.

With the use of smart constructors generated by `mkConstructors` and the updated definitions for `partition` and `caseof`, we get the final implementation of our pattern matching framework. Listing 4.20 shows an updated version of the example presented in Listing 4.17, using the final versions of `partition` and `caseof`.

4.5 Haski implementation

This section presents some small examples to demonstrate how our presented pattern matching technique can be applied to Haski programs. As our pattern matching technique was implemented in the generic `E` expression language, reimplemeting it in the Haski compiler was a relatively straightforward process. There is one notable exception, which was the preservation of Haski's memory-bounded execution guarantees. This is discussed in detail in Section 5.3. The full Haski implementation can be found at a separate branch in the Haski GitHub repository[2].

Applying our technique to Haski programs produces very similar programs as for the `E` language, except with the `E` type substituted with the `Stream` type. Listing 4.21 shows a small function in a vein similar to the one shown in Section 4.3.2. It is possible to match on user-defined product types, whose constructors may have fields

[2]The latest commit as of this thesis is found at: `https://github.com/OctopiChalmers/haski/tree/d226b24e574fe3534e0afb1bc227b3b08020861c`

```
type ErrorCode = Stream Int
data Moisture  = Dry | Wet | Error ErrorCode
$(mkConstructors ''Moisture)
instance Partition Double Moisture where
    partition = [ \v -> (v >. 1, _Error 1)
                , \v -> (v <. 0, _Error 2)
                , \_ -> (v >. 0.2, _Wet)
                , \v -> (v <. 0.2, _Dry) ]

soilIsDry :: Stream Double -> Stream Int -> Haski (Stream Bool)
soilIsDry moisture = caseof moisture $ \case
    Dry        -> pure (val True)
    Wet        -> pure (val False)
    Error code -> pure (code ==. 1)
```

Listing 4.21: A small example function where our pattern matching technique is used with Haski types, visually very similar to the earlier `E` examples.

of non-finitely enumerable types, such as `Double`. Unlike Haski's existing `match` function, we do not need to enumerate the values of the scrutinee type. Rather, as with previous examples in the `E` language, we use language primitives such as `(>.)` to operate on such values. Because of this, using non-finitely enumerable types in this way does not require any special treatment, as long as appropriate language primitives exist. In the below example, we would need for `(>.)` and `(<.)` to be defined, with the type `Stream Double -> Stream Double -> Stream Bool`.

5

Discussion

The pattern matching framework presented in Chapter 4 provides new functionality over the existing pattern matching implementation in Haski. In this chapter, we evaluate the technique used and discuss its advantages and limitations, comparing them to that of to Haski's existing `match` function. We also discuss the importance of Haski's clock-calculus for ensuring that programs are run with bounded memory usage, and analyze whether our framework can be safely used without violating this property.

5.1 Comparison of `caseof` and `partition` to `match`

The goal of this project was to develop a technique for improving the pattern matching functionality in Haski, providing an alternative to Haski's existing `match` function. This section presents the main ways our pattern matching technique, using `caseof` and `partition`, compares to `match`.

5.1.1 Handling the staging problem

As mentioned in Section 2.3, the core problem we run into when attempting to embed pattern matching is that the inspection of the scrutinee of a pattern match needs to be delayed until the runtime of the target program, since the value of the scrutinee is known only then. "Target program" here refers to the binary compiled from the target code, i.e. C in the case of Haski. However, while the value of the scrutinee cannot be known until the target program is run, the same is not the case for its *type*, which is known as early as during the compilation of the host program.

The technique used with `match` exploits this fact to work around the staging problem. If the input type to the `match` function is known to be finitely enumerable, then it is possible to, at an earlier stage, apply the matching function to every possible value that the scrutinee could have, and get a mapping to every single output possible. This map can then easily be embedded into the target code in the form of a `switch`-statement. In other words, the matching function is used to pre-compute every possible results of the `match`-expression. Because only the results of this pre-computation needs to be embedded, the matching function can be written using any normal Haskell syntax, including pattern matching on the scrutinee.

The `caseof` and `partition` functions employs a different technique, albeit with some similarities, and can be separated into two parts: *branching* and *construction*.

An intuitive way to embed pattern matching is to essentially translate a case expression into a multi-way `if-else`-chain (or, as with `match`, a `switch`-statement). To do this, it is necessary to somehow capture the *branching* logic that is implicitly performed in a case analysis. In the branches returned by `partition`, this is defined in the first half of the returned tuple as a predicate on the scrutinee; "do we select this branch or not?"

However, actual pattern matching, as described by the official Haskell report, is more complicated than a simple function from the input type to a boolean value [13]. Rather than attempt to model the actual pattern matching mechanism, we "cheat" by making the scrutinee analysis *not* directly correlate to the patterns accepted by the user-provided case expression. Instead, the analysis of the scrutinee only decides which value to *construct* of a *second* type; this second type is the actual type that the user-provided case expression matches on. Note that this second type can be the same type as the input, which can be useful in certain situations, such as the one described in Section 5.1.4.

In other words, we never actually pattern match on the scrutinee, only on the possible constructed values defined in a corresponding `Partition` instance. Similar to the technique used in `match` we can then apply the matching function to all branches defined in `partition` to get the a similar "mapping" from scrutinee type to output type.

5.1.1.1 Non-finitely enumerable types

The two techniques have a point in common, in that they create a mapping from scrutinee to output that can be easily embedded into the target code. The main difference is that the map generated by `caseof` does not map *exact* scrutinee values to corresponding output values. Rather, it is a map from *predicates* on the scrutinee to output values. In the former case, the number of branches is equal to the number of distinct values the scrutinee type. In the latter case, the number of branches can be arbitrary and is defined by the user in a `Partition` instance.

This is what avoids incompatibility with types that are not finitely enumerable. Mapping every distinct element of the input type naturally requires the type to be finite. To automatically generate all distinct values so that the matching function can be applied, requires the type to be enumerable. Contrastingly, using `caseof` only requires that, for every element of the input type, at least one predicate holds true. This gives flexibility to the user, who can define more or fewer predicates to suit the situation, and makes it possible to cover an infinite input space with a finite number of cases.

For example, determining whether a floating-point number is positive or negative only requires two branches, even though the `Double` type is non-finite:

```
data Sig = Pos | Neg
$(mkConstructors ''Sig)

instance Partition Double Sig where
    partition = [ \v -> (v >=. 0, _Pos)
                , \v -> (v  <. 0, _Neg) ]
```

Again, it should be noted that pattern matching using this technique does not actually pattern match on the scrutinee, but on the second type of the relevant `Partition` instance (`Sig` in the example above). However, it still allows for some additional use cases that are not possible with `match`. Consider the following type for example:

```
data Result = Success (Stream Double) | Failure
```

Essentially a specialized `Maybe` type, this type can be constructed in a `Partition` instance, and `caseof` can be used to perform pattern matching on the `Result` type. Using `match`, pattern matching on a value of `Result` is not possible, as the field of the `Success` constructor is not finite nor enumerable.

5.1.2 Representation of ADTs and patterns

The technique used with `caseof` carries an interesting property: while the user can define their own types to pattern match on, the pattern matching is "evaluated away" during the Haskell runtime when the program AST is constructed. Essentially, the type being matched on serves as an intermediate representation during program construction, and its definition or structure does not need to be embedded into the target code. For example, the watering application from Section 4.3.2 defines the `SensorData` type, which is used for pattern matching in the `needsWatering` function. However, when the program is compiled by running the Haskell binary, the definition of `SensorData` is not embedded into the resulting program, only the expressions resulting from evaluating the pattern match (which, again, is performed during the Haskell runtime). This also applies to the notion of patterns themselves, which currently do not require an internal representation either.

This is noteworthy as it greatly simplifies the internals of the EDSL's core data type (`E`, `Stream`). Techniques for representating Haskell's ADTs from within the language itself do exist, for example by using generic programming techniques [19, 20]. Template Haskell could also be a useful tool for this task, as it necessarily provides representations of all Haskell language constructs in the form of plain ADTs, albeit in a verbose fashion. Implementations using these techniques could resolve some limitations inherent to the current technique (discussed further in Section 5.2) or lessen the required boilerplate for the user, at the cost of potentially increased implementation complexity of the core EDSL type. This increase could negatively impact the ease of using this technique in other projects. The current technique

shows some promise of being relatively easy to integrate into existing projects, as a functioning Haski implementation was performed with few modifications to the existing code.

Nonetheless, difficulty of internal implementation does not concern the *user* of an EDSL much, to whom rich functionality would likely be of bigger importance. Therefore, more thorough ways of embedding patterns and ADTs could be well be worth the (potentially) additional implementation work, and has been explored in recent research [21].

5.1.3 User-defined types and boilerplate

Pattern matching with `caseof` on user-defined types does not require any special treatment where `caseof` is applied. There are, however, some limitations on the structure of any types to be matched on with `caseof`:

- In order to use the `mkConstructors` Template Haskell function, only simpler style data declarations are allowed. Currently, this includes "plain" `data` declarations and `newtype` declarations, but excludes declarations with type variables (such as `Maybe a`), infix data constructors, record constructors, and more.

- An embedded value of type `Stream a` consists of an underlying syntax tree representing an embedded expression with type `a`. This expression cannot easily be "un-embedded", as defining such a function of type `Stream a -> a` would require fully evaluating the embedded expression, which may not be possible.

  The user-defined functions given by `partition` take embedded values as input (i.e. `Stream Int` rather than `Int`) and can use them to construct new values of a second type. Because we cannot un-embed the inputs, the constructor fields of the constructed type often also need to be of the embedding type (`Stream`), to be able to make use of the input value. Section 5.2.1 discusses the reason and implications of this limitation in more detail.

Even with these limitations, this can still allow for a wide range of domain modeling in the style of the example from Section 4.3. Furthermore, the syntax used with the `caseof` function is similar to a normal Haskell case expression, in particular when used together with the `LambdaCase` extension. This serves to make the code performing the main program logic clean, as the boilerplate code can be separated and defined elsewhere. Compared to `match`, the syntax is almost identical, other than the name of the function.

There is a difference in boilerplate code, however. With `caseof`, it is necessary to write `Partition` instances and define the branches returned by `partition`. This could be considered an acceptable amount, especially if the instance is performing meaningful conversion logic as well, such as with the watering program from Section 4.3.

With `match`, the boilerplate mainly consists of `Bounded` and `Enum` class instances for scrutinee types. For some built-in types such as `Int8`, these instances are already defined. This is not the case for user-defined types, and Haskell is quite restrictive regarding the deriving of these instances. Specifically, the `deriving` keyword cannot be used to derive `Enum` and `Bounded` instances for types with non-nullary constructors. For instance, consider the following, taken from an example in [6]:

```
data Req  = Read | Write Bool
data Resp = Ok   | Val Bool
```

The `Req` and `Resp` types are isomorphic to `Maybe Bool` and only have three possible values each. However, because the necessary `Bounded` and `Enum` instances cannot be easily derived due to the non-nullary constructors `Write` and `Val`, they instead need to be defined manually:

```
instance Bounded Req where
    minBound = Read
    maxBound = Write True

instance Enum Req where
    toEnum 0 = Read
    toEnum 1 = Write False
    toEnum 2 = Write True
    fromEnum Read          = 0
    fromEnum (Write False) = 1
    fromEnum (Write True)  = 2

{- ... -}
```

It is likely, however, that it would be relatively straightforward to automatically generate these instances with Template Haskell, similar to that of `mkConstructors`. This would reduce necessary boilerplate written by the user to a minimal level, even if it introduces a dependency on Template Haskell.

5.1.4 Using `caseof` for `match`-like pattern matching

Thus far, there is one apparent missing feature from `caseof`-style pattern matching, compared to `match`. Consider the following simple example that uses `match`:

```
grow :: Stream Int8 -> Haski (Stream Int8)
grow ns = ns `match` \case
    0 -> 0
    n | n < 0 -> val (n - 1)
      | n > 0 -> val (n + 1)
```

Notably, this uses normal Haskell syntax to match on a literal `0` in the first case

and perform branching logic using guards in the second case. How would this be implemented using `caseof` instead? One option would be to create a new data type with a separate constructor for each branch, and then define a `Partition` instance from `Int8` to this new type:

```
data Sig = Zero | Neg | Pos
$(mkConstructors ''Sig)

instance Int8 Sig where
    partition = [ \v -> (v ==. 0, _Zero)
                , \v -> (v <.  0, _Neg)
                , \v -> (v >.  0, _Pos) ]

grow2 :: Stream Int8 -> Haski (Stream Int8)
grow2 ns = ns `caseof` \case
    Zero -> pure 0
    Neg  -> pure (ns - 1)
    Pos  -> pure (ns + 1)
```

This would work, but it is a lot of effort for a very simple task. Furthermore, while separating some logic into the `Partition` instance can be useful in some situations, it becomes an unwelcome cognitive overhead in this example. It would be helpful to have the option to use normal Haskell syntax with `caseof` as well.

Because we want to match on values of the same type as the input, we effectively want to bypass the conversion of the input normally performed by `partition`. One idea is to implement an "identity" instance `Partition Int8 Int8`, with a single branch in `partition` whose condition always returns `True` and returns the input value:

```
instance Partition Int8 Int8 where
    partition = [\v -> (val True, pure v)]

- Couldn't match type Stream Int8
-                with Int8
- Expected type: Haski Int8
-   Actual type: Haski (Stream Int8)
```

Unfortunately, this implementation does not type check; `pure v` is not of the expected type `Haski Int8`:

```
partition :: [Stream Int8 -> (Stream Bool, Haski Int8)]
v         :: Stream Int8
pure v    :: Haski (Stream Int8)
```

As mentioned earlier, we cannot easily un-embed a value, meaning we do not have an easy way to convert `v` from type `Stream Int8` to plain `Int8`. Instead, we could

try to change the "return" type of the `Partition` instance:

```
{-# LANGUAGE FlexibleInstances #-}
instance Partition Int8 (Stream Int8) where
    partition = [\v -> (val True, pure v)]
```

This implementation type checks! Unfortunately, it turns out that this `Partition` instance cannot be used to implement a `caseof`-version of the earlier `grow` function:

```
- Does not compile!
grow3 :: Stream Int8 -> Haski (Stream Int8)
grow3 ns = ns `caseof` \case
    0 -> pure 0
    n | n <. 0 -> pure (n - 1)
      | n >. 0 -> pure (n + 1)
```

Both the literal `0` pattern and the guarded `n | n ...` pattern are erroneous, for similar reasons:

- In order to determine whether the literal `0` is a matching pattern, it must be compared for equality with `ns`. This requires an `Eq` class instance for the `Stream` type, which is problematic to define for the same reason that `Stream`-embedded values cannot be easily un-embedded.
- The `n` in the `n | n ...` pattern is a value of type `Stream Int8`, meaning that comparison operators `<.` and `>.` have return type `Stream Bool` when fully applied. This is not compatible with Haskell's guards (`|`), which require the conditional expressions to be of the non-embedded type `Bool`.

As matching literal patterns and using Haskell's built-in guard syntax limits us to using non-embedded values here, we do need to implement the earlier `Partition Int8 Int8` instance somehow. The challenge is to have the functions in `partition` return values of type `Int8`, even when the input is of type `Stream Int8`.

```
1 {-# LANGUAGE ScopedTypeVariables #-}
2 enumIdPartitions :: forall a . (Enum a, Bounded a, CType a)
3     => [Stream a -> (Stream Bool, Haski a)]
4 enumIdPartitions = map build [minBound .. maxBound]
5   where
6     build :: a -> Stream a -> (Stream Bool, Haski a)
7     build x = \v -> (v ==. val x, pure x)
```

Listing 5.1: The `enumIdPartitions` functions enumerates all values of the type `a` to create branches that cover the entire input space, similar to that of `match`.

A partial solution to this is possible by using the same enumeration trick used by `match`! We can define the function `enumIdPartitions` as shown in Listing 5.1.

The purpose of `enumIdPartitions` is to generate one branch for every value of the bounded, finitely enumerable type `a`. Each branch returns a constant, non-embedded value of type `a`, and the predicate on the input holds true only when the input is equal the (embedded) value of this constant.

(2–3) The type of `enumIdPartitions` is very similar to that of `partition`. The differences are the additional `Enum` and `Bounded` constraints, as well as the constructed values of each branch having the same type `a` as the input type.

(6–7) The helper function `build` constructs a branch. It takes a plain value of type `a` as its input, and uses equality to its *embedding* as the the condition for the branch, while the return value is the non-embedded value of the same type `a` as its input.

(4) The plain values of type `a` required for the `build` function can be generated because of the `Bounded` and `Enum` constraints on `a`. By mapping `build` over the list of all values of `a`, we get all possible branches, much like with `match`.

We do not transform a `Stream Int8` into an `Int8` for returning, but instead return one of the `Int8` that was generated by the `[minBound .. maxBound]` expression. To ensure that the generated value is correct we guard the branch behind an equality check, for which we also make use of the generated `Int8`.

By defining `partition = enumIdPartitions` for the `Partition Int8 Int8` instance, the original `grow` example successfully compiles and generates code similar to the following:

```
/* ... */
if (SCRUTINEE == -1) return -2;
if (SCRUTINEE == 0)  return 0;
if (SCRUTINEE == 1)  return 2;
/* ... */
```

Indeed, this looks almost identical to if we had used `match`, except with the `switch`-statement being replaced by a chain of `if`-statements. This carries the same advantages as the `match`-style pattern matching, the main one being that our matching function can match on non-embedded values. This is what allows the use of more normal Haskell syntax to match literal patterns and use the built-in guards in the `grow` example from earlier. However, this also does not improve on the `match`-style technique, and its main purpose is only to unify the two techniques into a single function name. As `enumIdPartitions` has the same constraint as `match`, of requiring the scrutinee to be of a finitely enumerable type, it is not compatible with any more types than `match`.

5.2 Shortcomings of `caseof` and `partition`

This section will highlight some areas where the pattern matching technique using `caseof` and `partition` is lacking, both with regards to functionality and ergonomics.

5.2.1 Usability issues with non-embedded constructor fields

In the examples in Chapter 4, all partition types used either had nullary constructors (with zero fields), or had fields of type `E a` for some type `a`, for example:

```
data Sig = Pos (E Double)  {- ... -}
data T = T1 (E Int)  {- ... -}
```

While this is not strictly necessary for using our pattern matching technique, it is difficult to write useful instances of `Partition` otherwise. This is because there is no way, as mentioned earlier, for the user to "unwrap" a value of type `E a` to get a value of type `a`. As a reminder, the type signature of `partition` is the following:

```
partition :: Partition a p => [E a -> (E Bool, Estate p)]
```

The returned functions take `E a` as input, meaning that constructors of the output type `p` can only make use of the input value in its embedded form, i.e. as a value of type `E a`. We can therefore see that if the `T1` constructor had the type `(Int -> T)` instead of `(E Int -> T)`, there would be no way to meaningfully use the input value.

Enforcing this structure on types may not be a particularly bad issue for user-defined partition types, as they can be designed to account for this, as has been done in examples so far. However, there can be problems if attempting to reuse non-user-defined types as partition types, when the types have non-nullary constructors, such as `Maybe` and its `Just` constructor. In particular, the automatic generation of smart constructors described in Section 4.4.3 currently does not work well with types that were not defined with the use of `caseof` and `partition` in mind.

To demonstrate the problem, we can inspect the code generated for the `Just` constructor when running `$(mkConstructors ''Maybe)`:

```
_Just :: a -> Estate (Maybe a)
_Just v0 = do tag0 <- newFieldTag (Just "a0")
              return (Just (tag0 v0))
```

Recall that the return type of `newFieldTag` is `(Maybe String -> Estate (E a -> E a))`, meaning that `tag0` has the type `(E a -> E a)`. As `v0` has type `a`, the application of `tag0 v0` does not type check; `mkConstructors` makes the assumption that all fields of partition types are of the `E a` form, while the type signature of `_Just` accepts *any* `a` (including non-`E` values).

It is possible to work around the issue to some degree by defining a separate Template Haskell function for some common cases. For example, we could define a `mkLiftedConstructors` function which behaves like `mkConstructors` except it also specifies the type of the generated functions to be compatible with the embedded language and the use of `partition`:

```
$(mkLiftedConstructors ''Maybe)
```

```
- Generated by mkLiftedConstructors
_Just :: E a -> Estate (Maybe (E a))
_Just v0 = do tag0 <- newFieldTag (Just "a0")
              return (Just (tag0 v0))
```

This workaround can be used for types where the fields of its constructors are entirely parameterized by type variables, since they do not require the definition of the type to change. Simply using a more specialized instance of it is sufficient, such as using `Maybe (E a)` instead of `Maybe a`. Unfortunately, it is a flimsy workaround, and while it works for types such as `Maybe` or `Either`, even a very simple type like `data I = I Int` remains a problem since we cannot implement a sensible function with type `(E Int -> I)` without changing the the definition of `I` to something like `data I = I (E Int)`.

In practice, this should not a major hindrance, since the entire purpose of partition types is to be defined by the user for use with `caseof`. However, it is indicative of the difficulty of working with embedded values in a fully natural way, and how this can cause our pattern matching embedding technique to become clumsy and restrictive at times, something that will become more noticeable in the following section.

5.2.2 Restrictions on the scrutinee type

The nature of `partition` having to use an embedded value as its input restricts not only the structure of partition types, but also places significant restrictions on the scrutinee type. Unlike the partition types which are used only during program construction, the scrutinee and any language primitives operating on it needs to be embedded into the program, and in turn, the generated code.

This imposes two significant (and related) limitations on possible scrutinee types. Firstly, because the scrutinee must be representable in the target language (must have a `CType` instance), there is currently no general support for scrutinee types defined with "normal" Haskell ADTs. In our examples and in the reference implementation, the supported scrutinee types consists of a selected number of built-in types that have obvious C representations, such as booleans and basic numeric types. To support scrutinees with types defined using ADTs, the framework must also support a representation of ADTs in the embedding.

Secondly, because we cannot inspect the embedded scrutinee in the `partition` function, any operations on it must be performed by language primitives in the EDSL. This effectively means that the embedded language itself needs to be extended for any new scrutinee types that it wishes to support. For example, `E` defines primitives for boolean logic (`EAnd`, `ENot`) and basic arithmetic (`EAdd`, `ENeg`). Similar to the previous point, however, the lack of a general representation of ADTs in the frame-

work limits the number of useful scrutinee types. Note, however, that the issue of not being able to inspect the scrutinee is twofold here: the value is opaque due to being embedded, but there is also the staging issue from Section 2.3 to account for!

These restrictions are fundamental to our embedding technique and are harder to ignore compared to those previously discussed in Section 5.2.1. Only being able to interact with the scrutinee through language primitives effectively limits us to a few common built-in, non-inductive types (numeric types, booleans). It is not clear if the current technique used can be generalized to work well for a wider range of scrutinee types, or if a fundamental re-design is necessary. This is an issue for our technique's usability for EDSLs in general, but may not matter in the context of Haski, as the data generated by embedded devices will take the form of bitwise-encoded streams of plain numbers.

5.2.3 No exhaustiveness guarantee

One useful property of the enumeration technique used by `match` is that it automatically detects if the matching function is partial. Since `match` will attempt to apply the matching function to every possible value of the scrutinee type, a non-total function will cause a runtime error to be raised when running the host program. Note that a runtime error in the host program effectively becomes a compilation error for the embedded language, unlike a runtime error in the target program. Furthermore, because the type of the scrutinee is a plain, non-embedded value, even the Haskell compiler can assist in detecting non-exhaustive patterns if pattern matching is used in the matching function.

Pattern matching with `caseof` and `partition` does not have the same safety guarantees. Non-exhaustive patterns in the matching function can still be detected by the Haskell compiler, but the same does not apply for the branching conditions defined in `partition` as embedded expressions. Because the branching conditions are defined as predicates on the scrutinee in the *embedded* language, they are opaque to the Haskell compiler. Furthermore, because the predicates are not evaluated until the runtime of the target program, i.e. C, it is not possible use the same technique as `match` to produce a Haskell runtime error on non-exhaustive cases.

In other words, this issue cannot be guaranteed to be caught during compilation, neither by the Haskell compiler, nor the Haski compiler. Worse, manually checking that the conditions in a `partition` definition are exhaustive is more challenging than normal, since one would need to understand the syntax of the embedded language, rather than plain Haskell. This could be significant downside, since one clear goal of Haski is to produce safer C code and avoid typical bugs related to memory or undefined behavior. Having non-exhaustive branching conditions in `partition` could lead to none of the generated `if`-statements being run in the target code, and is a potential source of undefined behavior. Not being able to get a warning during the either compilation phase is therefore a clear issue with regards to Haski's goals.

One possible way to slightly mitigate the damage would be to automatically insert a "catch-all" code section at the end of the generated `if`-statements, which could

alert the user and immediately exit the program:

```
int f3(int x){
    if (x > 0) { return 100; }
    if (x < 0) { return 99; }
    /* If none of the branches fire: */
    printf("Non-exhaustive conditions in function `f3`\n");
    exit(1); }
```

While this would still cause a runtime error in the target program, it would at least ensure that the user is alerted if this particular bug is triggered and might be preferable, even if not ideal.

5.2.4 Potentially unintuitive precedence of patterns

Normally when pattern matching in a case expression, patterns are evaluated in order, starting from the top. This is still technically true in the matching function when using `caseof`, but is misleading, since the actual branching logic depends on the definitions and **order** of the branches in the corresponding `partition` definition. The order in which branches will appear in the generated C code depends only on the order in which they are defined in `partition`, **not** their order in the matching function when applying `caseof`.

```
data Size = Small | Medium | Large
$(mkConstructors ''Size)

instance Partition Int Size where
    partition = [ \v -> (v >. 10, _Medium)
                , \v -> (v >. 50, _Large)
                , \v -> (v >. 0,  _Small) ]

isLarge :: Stream Int -> Haski (Stream Int)
isLarge ns = ns `caseof` \case
    Large -> pure 1
    _     -> pure 0
```

```
int case_of_1(int arg) {
    int SCRUTINEE = arg;
    if (SCRUTINEE > 10) return 0;
    if (SCRUTINEE > 50) return 1;
    if (SCRUTINEE > 0)  return 0; }
```

Listing 5.2: Because the ordering of the branching logic in the generated C code depends on the definition of `partition`, the ordering of patterns in the `isLarge` function does not matter, and may become misleading. The generated `case_of_1` function shows the actual ordering of the branching conditions.

For an example of how this could become confusing, consider the code in Listing 5.2. Imagine that `isLarge` is given a constant stream of `60` as input. Looking only at the definition of `isLarge` and the corresponding entry in the list given by `partition`, the user might reasonably expect `isLarge` to return `1`. However, this will not be the case, since the `Medium` branch occurs before the `Large` branch in the definition of `partition`. The generated function corresponding to the `caseof` application in `isLarge` will therefore look like that of `case_of_1`.

The second `if`-statement will not be reached, since the `> 10` condition will always be true and occurs first in the ordering. A user familiar with Haskell but new to Haski may not immediately understand the cause of this kind of bug. This is at odds with one of the goals of this project, to provide an intuitive and familiar interface to users already comfortable with Haskell. While the usage of `caseof` and `partition` may be intuitive and understood, a detail like this could still cause problems for a user and be hard to track down without trial and error.

This is a difficult problem to resolve, since the ordering of patterns *do* normally matter when pattern matching in Haskell and thus, the order-sensitive behavior of `caseof` and `partition` may be necessary. A user familiar with Haskell will likely be aware that ordering of patterns matter, but miss that the ordering matters in `partition`, and not in the application of `caseof`. This would likely only a problem for each user once, until they become aware of this quirk, but it would still be desirable to not have this potential usability problem at all. Writing clear and easy-to-find documentation regarding this property of `caseof` could be a way to lower the risks of this behavior causing issues, without requiring a re-design of the framework itself.

5.3 Safe usage of `caseof` with Haski's clock calculus

Haski's design is adopted by from the synchronous programming language Lustre. Functions operate on streams of data, which can be intuitively viewed as infinite lists. Streams do not necessarily output values at the same rate; they can be slower or faster relative to each other, and output values at different times. The frequency of a stream can be altered by functions applied to it, such as Lustre's sampling operator `when`, which Haski also implements:

```
when :: Stream a -> (Stream b, b) -> Stream a
```

The expression `when s1 (s2, x)` acts as a filter on the stream `s1`, producing the value of `s1` only when the value of `s2` is equal to `x`, yielding a stream that is slower than `s1` [6].

This introduces an additional aspect to functions that operate on streams: functions may require that not only the value types of their arguments match, but also that the timings of the streams are correct. The *clock* of a stream can be thought of as a

boolean sequence of values paired with the stream, which indicates *when* a stream is present [11]. For example, operations such as multiplication or addition on streams are applied pointwise to its operands, and expect both its operands to have the same clock. That is, that they always output values at the same points in time.

```
{-# LANGUAGE RecursiveDo #-}

alts :: Haski (Stream Bool)
alts = mdo
    t <- True `fby` f
    f <- False `fby` t
    return t

halve :: Stream Int -> Haski (Stream Int)
halve nums = mdo
    bools <- alts
    letDef (nums `when` (bools, True))

bad :: Stream Int -> Haski (Stream Int)
bad = node "bad" $ \ns -> mdo
    slow <- halve nums
    letDef (ns + slow)
```

Listing 5.3: An ill-clocked program. In the definition of `bad`, pointwise addition is performed between two streams with different clocks, which is disallowed.

Clock calculus is the analysis of the clocks of streams to ensure that they are correct with regards to functions operating on them. This analysis is done to ensure that the program will be able to execute synchronously [22] and Haski relies on it to guarantee memory bound execution of its generated programs. To demonstrate how a problem of unbounded memory usage can occur otherwise, we can observe the example in Listing 5.3. The analysis of this example, as well as the visualization of clocks in Table 5.1, is heavily based on the motivation for clock checking in [11].

- The `alts` function defines a stream of alternating boolean values (True, False, True, ...). The `RecursiveDo` language extension is used to define the recursive monadic bindings for `t` and `f`.
- The `slow` function defines a function on integer streams, returning a stream of half speed by using the `when` function, outputting only when the value of `alts` is `True`.
- The `bad` function is a *node* definition, which will be compiled to a Haskell-embedded Lustre node. In Lustre, nodes act as abstractions over stream expressions, and can take externally provided streams as input [6].
- The `letDef` function defines the representation of a Lustre equation, which make up Lustre programs [7]. In Haski, applying `letDef` to an expression

creates a binding to a fresh variable for that expression, which is kept track of in the `Haski` monad. For the purposes of this report, it is only important to know that Haski's clock-checking is dependent on the variable bindings generated by `letDef`. Expressions can bypass clock-checking if they are not bound using `letDef` (or a function using `letDef`, such as `fby`). Enforcement of `letDef` to avoid clock-checking bypassing is *not* done by the compiler, meaning that it is possible to write and compile ill-clocked Haski programs. Such programs do not have explicitly defined behavior. Some expressions are clearly safe even when not explicitly clock-checked, such as constants. Others, such as `when`, may alter the clock of a stream, and need to be checked. As [6] does not explicitly state minimal "safe" usage of `letDef` (in terms of clock-checking), our examples will be using `letDef` liberally, possibly including redundant or unneeded applications.

Time	0	1	2	3	4	...
alts	T	F	T	F	T	...
ns	ns_0	ns_1	ns_2	ns_3	ns_4	...
slow	ns_0		ns_2		ns_4	...
ns + slow	$ns_0 + ns_0$		$ns_1 + ns_2$		$ns_2 + ns_4$	...

Table 5.1: Applying pointwise addition on the differently clocked `ns` and `slow` results in the faster stream `ns` buffering values in an unbounded queue. If run continuously, this queue will grow and eventually overflow.

Assume that `bad` is given the stream of natural numbers (0, 1, 2, ...) as input. Table 5.1 shows the values of the different streams at different points in time. The stream expression `ns + slow` applies pointwise addition to two streams with different clocks. If no values are lost, this means that buffering will need to performed for the values in the faster stream `ns`. This buffer will grow without bounds if the program is run continuously, eventually leading to a memory overflow. By rejecting ill-clocked programs such as this, Haski guarantees execution with bounded memory consumption.

As part of Haski's compilation process, the clocks of stream expressions are inferred and checked for correctness. A clock mismatch will result in an error when compiling the Haski program. When implementing our pattern matching technique, we need to ensure that we do not introduce new ways for clock-incorrect programs to pass compilation, as this would compromise the guarantee of memory-bounded execution. We will not formally prove that the implementation of `caseof` in Haski is clock-safe, and will instead only provide the following informal reasoning:

Clock checking becomes relevant for functions which depend on, or affect the speed of streams (such as `when`), or for functions which combines multiple differently-clocked streams into one. Notably, clock checking does not depend on the values of a stream, only their timings. While functions like `when` can affect the clock of a stream, comparing the clocks themselves between two streams is independent of their values. In other words, how a stream function can affect the clock checking of a program depends only on whether that function can affect the clocks of the streams it operates on, whether it is *clock preserving* or not.

The following function `square` is an example of a clock-preserving function, as it will always produce a stream with the same clock as its input:

```
square :: Stream Int -> Stream Int
square ns = ns * ns
```

Drawing from the above reasoning, we can see that clock-preserving stream functions act as "identity" functions, clock-wise. These are functions which take a single stream as input and return a single stream as output, where both the input and output have the same clocks. A function with these properties acts as an identity function on *clocks* since it cannot affect the clock of a program, wherever it is used. One can loosely compare it to the Haskell `id` function, which is the identity function on Haskell *values*.

Unfortunately, `caseof` is *not* clock-preserving by definition, since the matching function applied to the scrutinee may use clock-altering operations such as `when`. Therefore, we will instead define explicit clock-checking rules for `caseof` to force clock-preservation, which would force the matching function to be clock-preserving as well.

We define the clock checking of `caseof` in the following way. As described in Chapter 4, applying `caseof` creates a construct consisting of the following components:

- The input scrutinee, an expression of type `Stream a`
- A list of matches, each consisting of a condition expression of type `Stream Bool`, and a body expression of type `Stream b`, where `b` is the output type of the `caseof`.

We denote the scrutinee as *scrut*, the condition of the nth match as $cond_n$, and the body of the nth match as $body_n$. We denote an expression e with clock c as e^c. To have a well-clocked `caseof` construct with n matches, we enforce $scrut^x$ and $({cond_n}^x, {body_n}^x)$ for all condition expressions and body expressions $\{(cond_0, body_0)$, $(cond_1, cond_1)$, ..., $(cond_n, body_n)\}$, where x is some clock. In other words, we require that all condition expressions and body expressions have the same clock as the scrutinee.

By clock-checking `caseof` according to these rules, we can guarantee that applications of `caseof` will be clock-preserving; the input and each possible output of a `caseof` construct will all have the same clock. The previously discussed `letDef` requirement for clock-checking still applies; as with other Haski expressions, they are not guaranteed to be correctly clock-checked at all unless bound with `letDef`. It is possible that these rules are overly conservative since they aim to ensure clock-correctness for `caseof` in a vacuum, independent of the rest of the program. For example, different branches of a single `caseof` construct might not need to have equal clocks if the rest of the program is designed to compensate for this elsewhere. However, determining more precisely how to efficiently clock check `caseof` was outside of the scope of this project, and is left as potential future work.

6

Related work

This chapter discusses related work in the fields related to this thesis, such as previous work or research that has been applied to our framework design. Some possible techniques for DSL embedding that were not explored by this work are discussed as well. As Haskell has been used frequently as a host language for EDSLs, some notable projects are covered, and how their characteristics compare to or could be applicable to Haski.

6.1 Views and pattern synonyms

The pattern matching framework presented in Chapter 4 uses a technique that is heavily reminiscent of the *views* mechanism proposed by Wadler [12]. Indeed, many aspects of the use of `caseof` and `Partition` instances have clear counterparts in Wadler's paper, as well as GHC's later *pattern synonyms*.

In a functional language with ADTs, one common way of achieving data abstraction comes in the form of abstract types: types that do not expose their implementation or internal structure. This design is at odds with pattern matching, which depends on the structure of a type being visible in the form of its constructors. Views are presented as a way to improve how these concepts can coexist. By defining a *view* on a type, it is possible to pattern match on different constructors than the ones defined for the viewed type. A view on a type can concretely be thought of as a separate *viewer* type to the type that is being *viewed*, with accompanying "in"- and "out"-functions being defined to translate values between the two types.

An example of a view, taken from the paper and adapted for Haskell, is shown in Listing 6.1. When using views, `peanoIn` is automatically applied when `Zero` and `Succ` appear as patterns in place of an `Int`, such as on the left-hand side of an equation. The `peanoOut` function is the inverse of `peanoIn`, and is automatically applied when `Zero` and `Succ` are used as expressions in place of `Int`s, such as on the right-hand side of an equation. It is possible to define multiple views on the same type and by doing so, a single underlying type can effectively get multiple interfaces for pattern matching and value construction.

By default in Haskell, the patterns `Zero` and `Succ` can only be matched on after explicitly applying `peanoIn` and pattern matching on its result:

```
data Peano = Zero | Succ Int

peanoIn :: Int -> Peano
peanoIn n | n == 0 = Zero
          | n > 0  = Succ (n - 1)

peanoOut :: Peano -> Int
peanoOut Zero     = 0
peanoOut (Succ n) = n + 1
```

Listing 6.1: The type `Peano` is a viewer type, with `Int` being the viewed type. The `peanoIn` and `peanoOut` functions are used to translate `Int` values to and from the viewer type.

```
decr :: Int -> Maybe Int
decr n = case peanoIn n of
    Zero   -> Nothing
    Succ x -> Just x
```

By applying `peanoIn`, the `Peano` type can be used as an intermediate representation, solely for the purpose of pattern matching. However, having to explicitly apply the translation function is a bit clunky and lessens the usefulness of this design pattern, and does not reflect the convenience described by Wadler's proposal.

An improvement can be had with the use of GHC's `ViewPatterns` language extension, which allows expressions to occur in patterns and for computations to be performed during matching:

```
{-# LANGUAGE ViewPatterns #-}
decr (peanoIn -> Zero)   = Nothing
decr (peanoIn -> Succ x) = Just x
```

The `peanoIn` function will be applied to the argument of `decr`, the result of which will be compared to the `Zero` and `Succ` patterns. This syntax is more ergonomic, especially when the viewed type occurs deeply nested inside another pattern. The explicit inclusion of the `peanoIn` function is still required, however. Furthermore, this application of views is unidirectional: unlike the views described by Wadler, it is not possible for a written `Succ 3` to be implicitly translated by `peanoOut` into an `Int` value `4`.

Pattern synonyms were implemented in Haskell as of GHC version 7.8.1, as the `PatternSynonyms` language extension [23]. A dedicated report on the technique was later presented in 2016 [18]. With pattern synonyms, we can define both unidirectional and bidirectional symbolic patterns that can perform computation in either direction, allowing us to fully implement bidirectional views.

Listing 6.2 shows `Zero` and `Succ` being implemented as *explicitly bidirectional* pat-

```
{-# LANGUAGE PatternSynonyms #-}
{-# LANGUAGE ViewPatterns #-}
pattern Zero <- (view -> Left n)
  where Zero = 0
pattern Succ x <- (view -> Right x)
  where Succ n = n + 1

view :: Int -> Either Int Int
view n | n == 0 = Left n
       | n > 0  = Right (n - 1)
```

Listing 6.2: Patterns `Zero` and `Succ` are implemented as pattern synonyms, without an explicit viewer type. Each pattern synonym has two definitions, one for the pattern definition and one for value construction, which correspond to the "in"- and "out"-functions of views, respectively.

tern synonyms, allowing each direction of the translation to be defined separately. Pattern synonyms forgo the need for an explicit viewer type, and the "constructors" are instead defined using the `pattern` keyword. The first line of each pattern synonym definition uses the helper function `view` in a view pattern in a way that corresponds to the earlier `peanoIn`. The second line defines a *builder* separately, which corresponds to the behavior of the `peanoOut` function. Using these pattern synonyms, the `decr` function can be defined in a very clean way:

```
decr Zero     = Nothing
decr (Succ n) = Just n
```

While the upside of using views and patterns is not as obvious in this small and contrived example, being able to separately define the viewing and building of the viewed type can be very powerful for separating the implementation and interface of a data structure.

6.1.1 Comparison to `caseof` and `Partition`

It turns out that this separation of internal representation and public interface is something we want when implementing a pattern matching embedding as well, as discussed in Section 3.1. Looking closer at the technique in Chapter 4, we can see that it very strongly resembles Wadler's views-mechanism.

When used with `caseof`, `Partition` instances function very similarly to unidirectional views. A viewer type provides what is essentially a different interpretation of the viewed type, in the form of different constructors and automatic translation from the viewed type to the viewer type. This is translation is precisely the task of the `partition` function, which corresponds to the "in"-function of views. In a `Partition a p` instance head, the input type (viewed) `a` corresponds to the viewed type, and the partition type (viewer) `p` corresponds to the viewer type. Just like multiple views can be defined for the same type, there can be multiple instances

of `Partition`, with different partition types `p` providing different interpretations of the input type.

With all these similarities, and with pattern synonyms being able to implement views as proposed by Wadler, the obvious question is why the pattern matching technique in Section 3.1 was not implemented using pattern synonyms. It seems plausible that we could avoid the cognitive overhead of type classes if we could define the functionality of `partition` in pattern synonyms directly. It turns out that there are subtle differences from the example in the previous section, of which staging problems is the main cause. Directly substituting pattern synonyms for the `Partition` class, in an otherwise equivalent implementation, ends up not working very well.

The current implementation embeds the branching logic of pattern matching by constructing an embedded predicate on the scrutinee (or more precisely, a symbolic variable representing the scrutinee), and by embedding an association between each predicate to the corresponding result. Crucially, no matching can be done on the scrutinee during the Haskell runtime, as the scrutinee is an embedded value that cannot be inspected.

By using pattern synonyms, the functions given by `partition` are spread across multiple `pattern` definitions, for which they serve as the view function (in Listing 6.2, it would take the of where `view` is applied). This is already somewhat less ideal, since there is no forced grouping of related viewing type constructors. However, the bigger obstacle is that the view functions of pattern synonyms *do* perform matching on the scrutinee, as is their purpose. This is problematic for us; intuitively, the conditional check on the scrutinee should be performed here, but due to the discussed opaqueness of embedded values, this is not possible. As such, there is no sensible way to decide whether to reject or accept a pattern during the Haskell runtime, when `caseof` is applied. This matters, as it becomes more cumbersome to iteratively "apply" every pattern synonym in the matching function given to `caseof`, since the first pattern listed will always (or never) match.

Furthermore, the embedded predicate is part of the returned value from the functions given by `partition`, which is now the view function. To access the predicate outside the pattern synonym definition, such as inside the definition of `caseof`, the view function would need to be called separately anyway. To generalize this process over different types would end up requiring type classes regardless, mostly defeating the purpose.

What this points to is that our technique is fundamentally dependent on the use of type classes and their functionality can (perhaps obviously) not be directly replaced by pattern synonyms. Moreover, the main selling point of pattern synonyms, being able to perform computations as the scrutinee is inspected, is made irrelevant due to having an embedded and opaque scrutinee.

6.1.2 Pattern synonyms for EDSL design in general

Because pattern synonyms allow for arbitrary computation in pattern matches, they provide one method for bridging large gaps between a user interface and an internal representation. This can a powerful tool in any situation where pattern matching would be a viable way of interacting with abstracted away data.

For example, pattern synonyms are used in Haskell's list-like `Seq` data type to allow convenient access to the first and last element of a structure. The `Seq` type abstracts over an underlying *finger tree* structure [24] and provides useful operations for general-purpose sequences. Interestingly, EDSLs with a deeply-embedded core (including Haski) often share a similar design, where the user interface consists of smart constructors and a shallow embedding layer with useful operations that evaluate to one of the core type primitives. The use of pattern synonyms to extend the user interface with additional methods of constructing and manipulating the core type warrants further exploration, even if directly embedding its semantics have been show to be difficult.

6.2 Alternative embedding techniques

Reiterating what was stated in Section 3.1, a core challenge of embedding pattern matching is determining how to represent it as a data structure, while also providing a user-friendly interface to the user. In an ideal world, we would like to be able to use Haskell's case expression syntax directly to represent embedded case expressions in Haski. Instead, the `E` type uses the `ECase` constructor to represent a case expression, but this is a partial representation that only encodes the possible *results* of a case expression, not the patterns or types themselves. This is why `partition` is used to pre-compute the results, avoiding the need to store representations of patterns or ADTs in the embedding, as discussed in Section 5.1.2. The result is that the pattern matching framework does not feel as properly integrated with the rest of the language, compared to "native" pattern matching.

It is likely that more ergonomic and capable solutions for EDSL pattern matching in Haskell can be found, given better techniques for representing embedded data and methods for translating data into the embedding. This section covers some related projects that presents techniques that are potentially useful for solving the challenge of pattern matching embedding, either directly or through useful techniques for data representation and inspection.

6.2.1 Using a GHC plugin to reify pattern matching

Recent work by Young, Grebe, and Gill [21, 25] proposes solutions to both the representation aspect and the translation aspect of the embedding process:

For the representation, an expression language defined in an GADT is used, with constructors for representing pattern matches on a more granular level. To represent Haskell's ADTs, types are given a type-level representation in terms of `Either`, `(,)`, `()`, and `Void`. While `Either` is used to model a choice of types (sum), `(,)` models

a combination of types (product). Finally, the type `()` and `Void` are used as a base cases. By nesting the applications of these type constructors, it is possible to model ADTs with an arbitrary number of constructors, each with an arbitrary number of fields.

Still, even with a sophisticated representation of types and patterns, there remains the difficulty of providing an ergonomic interface to the user that can translate input into the internal representation. To handle this challenge, [21] presents an accompanying *GHC Core plugin*. *Core* is the name of GHC's internal intermediate representation, and can be inspected and modified using compiler plugins [26]. This additional power is key to the presented plugin, which translates "marked" Haskell expressions into corresponding ones embedded into the EDSL's expression language.

For marking expressions, a function called `externalize` is provided with the following type signature, where `E` is the type of the expression language:

```
externalize :: a -> E a
```

```
x :: Either Char Int
x = Left 'a'

ex :: E Int
ex = externalize $ case x of
    Left c -> ord c
    Right i -> i

ex' :: E Int
ex' = CaseExp
    (LeftExp (CharLit 'a'))
    (SumMatchExp
        (OneProdMatch (Lam 1 (Ord (Var 1))))
        (OneSumMatch
            (OneProdMatch (Lam 2 (Var 2)))))
```

Listing 6.3: An example of translation performed by the compiler plugin. The case expression in `ex` is marked with `externalize` to be targeted. The marked expression will then be translated by the plugin into its EDSL representation which can be seen in `ex'`.

Marked expressions of type `a` are then translated by the plugin to their EDSL representation as embedded expressions of type `E a`. Listing 6.3 shows a simple example adapted from [21], demonstrating how the plugin translates a marked case expression into its internal representation. In this example, the scrutinee is represented by the `LeftExp` constructor, while the branches are contained within the `SumMatchExp` constructor. Once a program has been translated into this form, it can then be used by backends to generate code in various target languages.

Notably, the case expression in the `ex` can be written using the normal Haskell syntax, only requiring the `externalize` keyword as a wrapper. This makes for very

intuitive syntax and is made possible by the compiler plugin, by providing a first-class data representation of otherwise opaque built-in language constructs. This allows for constructs like case expressions to be inspected rather than evaluated, from within Haskell itself.

A possible downside of this technique is that the use of a compiler plugin ties the implementation closer to the particular compiler implementation than it would be otherwise. For Haski, this could be a step back for portability, as it currently does not make use of any compiler plugins. While Haski does make use of several GHC-specific language extensions, the semantics of extensions are clearly specified, and are generally expected to be stable across newer GHC versions. Because a plugin exposes compiler internals, there is a higher probability of running into a backwards-incompatible API change due to a compiler version upgrade. It is therefore possible that a library using a compiler plugin may require more active maintenance to stay compatible with newer GHC versions, if so desired.

6.2.2 Template Haskell

Template Haskell is a language extension to GHC first proposed in 2002 [27] that allows a user to define compile-time functions that can be used to generate Haskell code before the rest of the program is compiled. As Template Haskell runs at compile-time and not during the Haskell runtime, it also enables metaprogramming capabilities through inspection of the source code. This capability is provided as a library included with GHC, and exposes types and constructors for interacting with Haskell language constructs at a lower level. Crucially, this includes representations for case expressions and patterns, meaning that Template Haskell provides both an encoding of Haskell constructs as data, as well as a means to translate regular source code into this encoding. Potentially this might be sufficient to provide both the representation and translation aspects of the embedding process, like [21].

Listing 6.4 shows a translation example similar to that of Listing 6.3. By using Template Haskell quotation syntax (`[|` and `|]`), the case expression defined in `ex` can be inspected as the syntax tree shown in `ex'`. `Exp` is the type of Template Haskell's representation of Haskell syntax, while the `Q` monad provides an environment and useful operations for working with the inspected code. Similar to the `E` type above, the representation provided by `Exp` can be used to embed pattern matching encodings into a core EDSL type, with the quotation serving a similar purpose to the earlier `externalize` function.

Unfortunately, the nature of how Template Haskell works imposes some limitations in how it can be used, which can impact user ergonomics negatively. Listing 6.5 shows one such example. Surrounding an expression with `$(` and `)` performs *splicing* on the expression. Splicing a value of type `Q Exp` will evaluate the encoded expression at compile-time and return Haskell code, before the rest of the program is compiled. Due to the process in which Template Haskell is processed in GHC, it is not possible to splice a top-level definition defined in the same module as the splice. In other words, the definition of `y` is not allowed here, and `ex` must instead be defined in a separate module and imported by the module defining `y`.

```
ex :: Q Exp
ex = [| case Left 'a' of
            Left c -> ord c
            Right n -> n
     |]

ex' :: Exp
ex' = CaseE
    (AppE (ConE Data.Either.Left) (LitE (CharL 'a')))
    [ Match
        (ConP Data.Either.Left [VarP 1])
        (NormalB (AppE (VarE GHC.Base.ord) (VarE 1)))
        []
    , Match
        (ConP Data.Either.Right [VarP 2])
        (NormalB (VarE 2))
        []]
```

Listing 6.4: Template Haskell quotation syntax can be used to inspect the case expression in `ex` as the form shown in `ex'`. The names of the translated variables have been altered for readability.

```
ex :: Q Exp
ex = [| case x of
            Left c -> ord c
            Right n -> n
     |]

y = $(ex)  - Disallowed splice!
```

Listing 6.5: A splice cannot be performed on a top-level expression defined in the same module as the splice. The definition of `y` is therefore disallowed and results in a compilation error.

Even though the use of Template Haskell can be unwieldy, it can be a relatively lightweight tool for some tasks that would only be possible with a compiler plugin otherwise. As already covered, the presented pattern matching embedding technique with `caseof` uses Template Haskell to generate smart constructors for use in `partition` definitions. The PanTHeon EDSL [28] uses metaprogramming in Template Haskell to hand-tune code optimizations without the use of a backend code generator.

The *quasi-quotation* extension to Template Haskell [29] increases its capabilities further, enabling customizable custom syntax for Haskell patterns and expressions. This can be used to allow special, more convenient syntax for particular constructs, if so desired. This is potentially very useful in a deeply-embedded EDSL, as it

provides a way to selectively work around the challenge of translating user input into constructs of the internal representation, as discussed in Section 3.1. The usefulness of quasi-quotation has not gone unnoticed, and has been used for various purposes in other EDSLs:

- Nikola [30] is an EDSL that compiles programs to run on GPUs by generating CUDA code. Quasi-quotation is used to allow the user to write CUDA code directly in a Haskell program, while maintaining a representation with proper types, rather than a much less robust string representation.

- Accelerate [31] is an EDSL that offers a high-level language for high-performance array computation by generating CUDA code, similar to Nikola. In Accelerate, quasi-quotation is used both to generate functions in the embedded language by quoting CUDA templates, as well as for exporting embedded functions as C functions for use by other languages [32].

- Ivory [33], an EDSL designed as a safer way to write embedded programs, uses quasi-quotation to define custom syntax for declaring and safely operating on bitwise data.

It could be worth exploring the usefulness of quasi-quotation in a solution for pattern matching embedding. In particular, it might allow for more complex internal representations of pattern matching if necessary, as the difficulty of translating to a more complex encoding could potentially be worked around by defining special syntax with quasi-quotation.

6.3 Early Haskell EDSLs for hardware

Using Haskell for embedding DSLs in hardware-related domains has been done for some time, likely due to the attractiveness of using a higher-level language in a typically low-level domain. Two early examples of Haskell-embedded DSLs are Hydra [34] and Lava [9], both in the domain of hardware design. Hydra is a language designed to aid in the teaching of computer architecture design, and provides a specification language for digital circuits. A hydra specification is immediately executable and is an example of an EDSL that does not have code generation for a different target language. The Lava EDSL, also a hardware specification language, showcases the technique of allowing multiple interpretations of the same circuit specifications. Through the use of monads and type classes, the same program can be overloaded to have different behavior depending on the interpretation used. The examples given in the paper includes in-Haskell simulation, similar to Hydra, as well as code generation for the VHDL hardware description language. Adding new interpretations is done by defining new monads, and is analogous to how a deeply-embedded core of an EDSL allows for different backends with different behaviors (code generators, interpreters etc.).

6.4 Feldspar

Feldspar [17] is an EDSL designed for describing digital signal processing algorithms, and is embedded in Haskell. It bears similarities to Haski, as it also uses a data-flow programming model and generates code targeting C. It also features a deeply-embedded core language like Haski, but makes great use of smart constructors, both to make on-the-fly optimizations during the construction of the core language program, but also to extend the language itself. The notable example given is the implementation of symbolic vectors, which do not exist in the core language, but only as separate types in a shallow embedding, which then gets translated to core language primitives. This technique was later presented separately [35], as a more general technique. The presence of a shallow embedding layer can be seen in the implementation of `caseof` as well; the pre-computation of branches as defined by `partition` is a processing step performed during the construction of the core language program.

6.5 Ivory

Ivory [33] is an EDSL with the primary goal of being a safer alternative to languages like C and C++ for low-level programming. Its type system is embedded into Haskell's, allowing for the GHC compiler to statically detect issues related to memory and many types of undefined behavior. This is while still providing an interface for low-level memory manipulation, making Ivory suitable for programming embedded devices.

Ivory shares Haski's main goal of providing a high-level way to program devices that require low-level operations, while having a minimal runtime and ensuring memory-safety. While Haski's Lustre-based architecture differs from that of Ivory, the report on Ivory brings up interesting problems and solutions in several areas that could be worth exploring for Haski as well. One such area is Ivory's implementation of a safe interface for bitwise manipulation of data, including common operations such as bitwise shifting and masking. Ivory implements a model of bitdata representation presented in [36], and defines custom quasi-quoters [29] to provide a convenient interface to the user. Having a similar target domain, the usability of Haski could greatly improve with the implementation of such techniques.

Another point of interest is how Ivory handles the issue of accurate error reporting in an embedded language. While it would be desirable to not get unexpected runtime errors in the generated code, this can be hard to guard against when the trigger condition is dynamic, such as a division-by-zero situation. In such cases, it should be be preferable to get a runtime error rather than undefined behavior, for example by an automatically inserted assertion check. However, while the error will manifest during the runtime of the target program, the programming fault is found in the embedded program, i.e. the code of the host language. Ideally, the error should point to the appropriate error source in the Haskell code, rather than in the C code. Ivory implements a compiler plugin to resolve this, as it can tap

into GHC's compilation pipeline and recover source information to be added to the internal representation. This plugin was made available as a separate package[1], but is currently unmaintained. For Haski, additional information from the source program could be very useful when debugging non-exhausting branches occurring in `partition` definitions, as discussed in Section 5.2.3.

6.6 Racket

As was brought up in Section 2.2, Haskell possesses some useful properties for an EDSL host language, most notably its type system. Additionally, it has a fairly lightweight and flexible syntax, with custom operators and overloading of functions through the use of type classes. When it comes to flexible syntax, however, it would be remiss not to highlight the Lisp family of languages and their macro systems. In the context of EDSL host languages in particular, Racket is a Lisp variant that is particularly interesting. Racket is a dialect of Scheme specifically designed for creating programming languages and has a very modular syntax system, even compared to other Lisp languages, and blurs the line between libraries and languages [37]. The Racket Manifesto [38] describes three main principles behind the design of the language, summarized below:

1. Racket heavily emphasizes the idea that programming problems often exist in the context of some domain that carries an associated domain-specific glossary, which should be captured in the programming language used. Or, put another way, the idea that "programming languages is about solving problems in the correct language".

2. Racket should be able to support multiple languages in the same host program, and components defined in different languages should be able to interoperate. Racket provides the means to define protection mechanisms for safe composition of components in different languages, while maintaining invariants enforced by each language. An example of such an invariant would be the type safety of components defined in the Typed Racket language variant, which allows Racket programs to be incrementally type annotated.

3. Racket captures aspects of programming that are normally external to the programming language itself, such as programmable modules and classes. The manifesto also uses the functionality of an integrated development environment (IDE) as an example, where Racket allows mechanisms for managing projects and resources to be internalized into Racket language constructs.

While Racket could be an interesting choice of host language for an EDSL in general, it was not of much practical interest for this thesis, which concerned Haskell specifically. Comparing the two languages, the most obvious difference would be regarding type safety. Racket does not have the same extensive type system that Haskell has built in. It is likely that safety properties can be added manually to some degree through extending the language (such as with Typed Racket), but Haskell's

[1] `https://hackage.haskell.org/package/ghc-srcspan-plugin`

type system has the advantage of being both feature-rich and easy to apply to an embedding "out of the box", such as by getting typed expressions in the embedded language via a GADT-defined deep embedding.

6.7 Other challenges in Haskell EDSLs

Embedding a language with a narrow purpose within a larger, more general language is an old idea. For example, Landin [39] was early in observing that many languages share the same fundamentals, with differences mainly concerning the particular domain that a language was designed for. Later, the likes of [40] and [41] have reiterated the idea of defining complete purpose-built languages in terms of a higher-level general-purpose one. Since then, in the context of Haskell EDSLs, some other notable challenges related to embedding have been observed, other than pattern matching specifically.

For one, pattern matching is only one of Haskell's mechanisms that is hard to embed. Recursion is also a fundamental technique used in Haskell programs and a potentially desirable inclusion in an embedded language. Specifically, as with pattern matching, what we actually want is to be able to have recursion on the embedded level be directly represented by Haskell-level recursion, and as with pattern matching, recursion is difficult to capture from within Haskell itself. Young, Gill, and Grebe's framework for pattern matching embedding also uses the same technique for reifying tail recursion and lambda abstractions [21], using a GHC Core plugin. Several other EDSLs substitute recursion for dedicated loop combinators instead, if such behavior is required [33, 17, 42].

Sharing is the idea that when equivalent expressions are used multiple times in a program, the result should be stored and shared between the locations, to avoid redundant computation. A compiler performs *implicit* sharing when it automatically finds and applies sharing for such occurrences. A language can allow the user to specify *explicit* sharing, such as with the `let` keyword in Haskell:

```
let x = sum [2, 3, 9]
in x * x
```

Sharing is not observable in Haskell, however; we cannot tell if two structurally equal values are shared, i.e. if `a = [1, 3]` and `b = [1, 3]` are, in fact, the same.

For a code-generating EDSL compiler that targets a low-level domain, the efficiency of the generated code can be highly relevant. The compiler for such an EDSL then, may need to be aware of sharing in the embedded program to be able to produce efficient programs. There have been various techniques presented to allow an EDSL compiler to gain some form of observable sharing.

In [43], a hardware description language embedded in Haskell presents the need for observed sharing. Because hardware circuits are cyclic graphs in nature, implementing them using Haskell's ADTs requires some degree of recursion in the

design. However, when inspecting such a structure, there is no observable difference between a cycle in the graph and an infinite tree. With the technique proposed by the authors, the addition of a reference type with pointer equality allows a degree of observable sharing to be introduced to nodes in the graph, which can allow the detection of cycles. [44] presents a different technique for observable sharing, using an `IO` function. This is argued to not be a major problem in practice, as the times where observable sharing is desired coincides with situations where I/O would be performed anyway. The technique is shown to be adaptable to work on other non-graph types in Haskell, as well as functions types. [45] discusses several implementations of observable sharing in Haskell, including the two previously mentioned. Also demonstrated is the implementation of a user interface to allow the user to define explicit sharing themselves in the embedded language, as well as potential pitfalls of re-using Haskell's `let` keyword for this purpose.

We ran into a similar problem of sharing in our pattern matching implementation (see Section 4.4.3), where the expressions in constructor fields were duplicated in the generated code. Our solution was an ad hoc implementation of common subexpression elimination, using tagged expressions. The main point of interest here was the automation of tag insertion, using the smart constructors generated with Template Haskell.

7

Conclusions and future work

We have designed a pattern matching framework for deeply-embedded Haskell EDSLs and implemented it in the Haski compiler. However, the design is a more general one and is not limited to that of Haski, nor does it depend on any of Haski's particular properties, other than it using a deeply-embedded design. The framework allows the user to define translations from a scrutinee (of a non-inductive type) to that of a second *partition* type, which can then be pattern matched on with a special `caseof` function. We have demonstrated the additional capabilities compared to Haski's existing method of pattern matching, such as improved support for non-finitely enumerable types, and we have shown that our framework can achieve at least feature parity while requiring a comparable amount of boilerplate from the user.

Our pattern matching framework is one implementation of pattern matching embedding into a Haskell-embedded language. For code-generating EDSLs, the problem of different compilation stages makes embedding pattern matching tricky. Unlike a normal Haskell program, a program in a deep embedding does not "run" during the Haskell runtime. Rather, the program is *constructed* at this time, to be further processed by a code generator in a later compilation stage. What this means is that any construct we wish to end up in the final program (such as pattern matching), must have some representation in our deep embedding.

Embedding Haskell's pattern matching is challenging, as its behavior cannot easily be reified from within the language. Our technique therefore "fakes" pattern matching, by using `Partition` type class instances to define alternative patterns for the scrutinee, similar to Haskell's pattern synonyms. Each instance of `Partition` defines branching conditions for each pattern alternative, whose value and condition is a function on the scrutinee. The application of `caseof` then uses the appropriate `Partition` instance to generate embedded expressions that correspond to chained conditional statements in the generated code.

There are several limitations to our pattern matching framework. For one, it does not allow for actual pattern matching on the scrutinee, only user-defined translations on it. The boilerplate required is non-trivial, even with the partial aid of Template Haskell. Having to define branches in `Partition` instances potentially obfuscates the code and bypasses some compiler guards against non-exhaustive patterns. Perhaps most problematic, however, is the fundamental restrictions placed on both the scrutinee types and partition types. Partition types need to use a certain structure to be compatible with our framework, a problem for existing types that were not de-

signed with this in mind. Scrutinee types are fundamentally limited to only support non-inductive types, but are realistically only usable with a handful of such types (numeric types, booleans). These types may actually be sufficient in the context of Haski, but may restrict our framework's usefulness in a general EDSL setting.

7.1 Future work

Looking forward, there are improvements to be made primarily in two areas:

1. As the framework was designed with the goals of Haski in mind, more domain-relevant features could be added to the language to further Haski's role as a safe alternative to C for IoT-devices. Such features would not need to build on the framework in this report, but could serve as separate language additions or stand-alone frameworks.
2. One could also focus further on the pattern matching aspect and explore ways to improve on our framework, or investigate alternative methods of pattern matching embedding.

For the first point, there is one significant feature addition that can be done to improve Haski's suitability as a language for programming IoT-devices: explicit support for working with bitdata. With safety as a high priority, it is only natural for the next step to be a dedicated DSL or interface for safely performing bitwise operations, such as masking and shifting. The implementation used in [33] would be an interesting example. Ideally, such an interface would make use of Haskell's type system to safeguard against accidental user errors, to the degree possible.

As for the second point, there is plenty of related research from which to get inspired. Our pattern matching framework can be considered clumsy and not well-integrated, in particular regarding the required boilerplate and the strict rules on the structure of types to use with `caseof`. While pattern synonyms and views have been shown to not be a direct replacement for our use of type classes, it would be well worth the time to explore how they can used to improve the ergonomics of our existing framework.

However, the stronger limitation lies in the restrictions on the scrutinee type, how our framework fails to support scrutinees with non-inductively defined types. Having a proper internal representation of Haskell's algebraic data types may be a significant step towards a better solution, but likely requires a fundamentally different approach than ours. For example, as has been discussed, using a compiler plugin to make normally opaque constructs such as case expressions inspectable, is an attractive prospect. Likewise so is more extensive use of Template Haskell: the quasi-quotation feature in particular has been popular amongst other successful Haskell EDSLs, and it would surely be of interest to explore the possibilities and downsides introduced with a more involved use of Template Haskell.

Bibliography

[1] E. Bertino and N. Islam, "Botnets and internet of things security," *Computer*, vol. 50, no. 2, pp. 76–79, 2017-02, ISSN: 1558-0814. DOI: 10.1109/MC.2017.62.

[2] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016-05, pp. 636–654. DOI: 10.1109/SP.2016.44.

[3] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16, New York, NY, USA: Association for Computing Machinery, 2016-05-30, pp. 461–472, ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897886.

[4] D. Goodin, "Record-breaking DDoS reportedly delivered by> 145k hacked cameras," *Ars Technica*, vol. 28, 2016.

[5] S. K. Bansal, "Linux worm targets internet-enabled home appliances to mine cryptocurrencies," *The Hacker News*, vol. 19, 2014.

[6] N. Valliappan, R. Krook, A. Russo, and K. Claessen, "Towards secure IoT programming in Haskell," in *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2020, New York, NY, USA: Association for Computing Machinery, 2020-08-27, pp. 136–150, ISBN: 978-1-4503-8050-8. DOI: 10.1145/3406088.3409027.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991-09, ISSN: 1558-2256. DOI: 10.1109/5.97300.

[8] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, "Copilot: Monitoring embedded systems," *Innovations in Systems and Software Engineering*, vol. 9, no. 4, pp. 235–255, 2013-12-01, ISSN: 1614-5054. DOI: 10.1007/s11334-013-0223-x.

[9] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," *ACM SIGPLAN Notices*, vol. 34, no. 1, pp. 174–184, 1998-09-29, ISSN: 0362-1340. DOI: 10.1145/291251.289440.

[10] J. Qian, J. Liu, X. Chen, and J. Sun, "Modeling and verification of zone controller: The SCADE experience in China's railway systems," in *2015 IEEE/ACM*

1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS), 2015-05, pp. 48–54. DOI: 10.1109/COUFLESS.2015.15.

[11] P. Caspi, G. Hamon, and M. Pouzet, "Synchronous functional programming: The Lucid Synchrone experiment," *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes*, pp. 28–41, 2008.

[12] P. Wadler, "Views: A way for pattern matching to cohabit with data abstraction," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '87, New York, NY, USA: Association for Computing Machinery, 1987-10-01, pp. 307–313, ISBN: 978-0-89791-215-0. DOI: 10.1145/41625.41653.

[13] S. Marlow *et al.* (2010). Haskell 2010 language report, [Online]. Available: https://www.haskell.org/onlinereport/haskell2010 (visited on 2021-06-25).

[14] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn, "Simple unification-based type inference for GADTs," *ACM SIGPLAN Notices*, vol. 41, no. 9, pp. 50–61, 2006-09-16, ISSN: 0362-1340. DOI: 10.1145/1160074.1159811.

[15] *The Glasgow Haskell Compiler*, version 8.6.5, 2020-04-07. [Online]. Available: https://www.haskell.org/ghc/ (visited on 2021-08-08).

[16] P. Graham, "The extensible language," in *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall, 1994, ch. 1, pp. 3–4, ISBN: 978-0-13-030552-7.

[17] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax, "Feldspar: A domain specific language for digital signal processing algorithms," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, 2010-07, pp. 169–178. DOI: 10.1109/MEMCOD.2010.5558637.

[18] M. Pickering, G. Érdi, S. Peyton Jones, and R. A. Eisenberg, "Pattern synonyms," in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016, New York, NY, USA: Association for Computing Machinery, 2016-09-08, pp. 80–91, ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976013.

[19] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh, "A generic deriving mechanism for Haskell," *ACM SIGPLAN Notices*, vol. 45, no. 11, pp. 37–48, 2010-09-30, ISSN: 0362-1340. DOI: 10.1145/2088456.1863529.

[20] E. de Vries and A. Löh, "True sums of products," in *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, ser. WGP '14, New York, NY, USA: Association for Computing Machinery, 2014-08-26, pp. 83–94, ISBN: 978-1-4503-3042-8. DOI: 10.1145/2633628.2633634.

[21] D. Young, M. Grebe, and A. Gill, "On adding pattern matching to Haskell-based deeply embedded domain specific languages," in *Practical Aspects of Declarative Languages*, J. F. Morales and D. Orchard, Eds., vol. 12548, Series Title: Lecture Notes in Computer Science, Cham: Springer International

Publishing, 2021, pp. 20–36, ISBN: 978-3-030-67437-3 978-3-030-67438-0. DOI: 10.1007/978-3-030-67438-0_2.

[22] P. Caspi, "Clocks in dataflow languages," *Theoretical Computer Science*, vol. 94, no. 1, pp. 125–140, 1992-03-02, ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90326-B.

[23] *GHC user's guide, Pattern synonyms*, version 9.0.1, GHC Team, 2021. [Online]. Available: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/pattern_synonyms.html (visited on 2021-08-06).

[24] R. Hinze and R. Paterson, "Finger trees: A simple general-purpose data structure," *Journal of Functional Programming*, vol. 16, no. 2, pp. 197–217, 2006-03, ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796805005769.

[25] M. Grebe, D. Young, and A. Gill, "Rewriting a shallow DSL using a GHC compiler extension," *ACM SIGPLAN Notices*, vol. 52, no. 12, pp. 246–258, 2017-10-23, ISSN: 0362-1340. DOI: 10.1145/3170492.3136048.

[26] *GHC user's guide, Extending and using GHC as a library*, version 9.0.1, GHC Team, 2021. [Online]. Available: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/extending_ghc.html (visited on 2021-08-06).

[27] T. Sheard and S. P. Jones, "Template meta-programming for Haskell," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '02, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, pp. 1–16, ISBN: 1581136056. DOI: 10.1145/581690.581691.

[28] S. Seefried, M. Chakravarty, and G. Keller, "Optimising embedded DSLs using Template Haskell," in *Generative Programming and Component Engineering*, G. Karsai and E. Visser, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, vol. 3286, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 186–205, ISBN: 978-3-540-23580-4 978-3-540-30175-2. DOI: 10.1007/978-3-540-30175-2_10.

[29] G. Mainland, "Why it's nice to be quoted: Quasiquoting for Haskell," in *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, ser. Haskell '07, New York, NY, USA: Association for Computing Machinery, 2007-09-30, pp. 73–82, ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291211.

[30] G. Mainland and G. Morrisett, "Nikola: Embedding compiled GPU functions in Haskell," in *Proceedings of the third ACM Haskell symposium on Haskell*, ser. Haskell '10, New York, NY, USA: Association for Computing Machinery, 2010-09-30, pp. 67–78, ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863533.

[31] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating Haskell array codes with multicore GPUs," in *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, ser. DAMP

'11, New York, NY, USA: Association for Computing Machinery, 2011-01-23, pp. 3–14, ISBN: 978-1-4503-0486-3. DOI: 10.1145/1926354.1926358.

[32] R. Clifton-Everest, T. L. McDonell, M. M. T. Chakravarty, and G. Keller, "Embedding foreign code," in *Practical Aspects of Declarative Languages*, M. Flatt and H.-F. Guo, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 136–151, ISBN: 978-3-319-04132-2. DOI: 10.1007/978-3-319-04132-2_10.

[33] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury, "Guilt free ivory," *ACM SIGPLAN Notices*, vol. 50, no. 12, pp. 189–200, 2015-08-30, ISSN: 0362-1340. DOI: 10.1145/2887747.2804318.

[34] J. O'Donnell, "Teaching functional circuit specification in Hydra," in *Functional Programming Languages in Education*, P. H. Hartel and R. Plasmeijer, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1995, pp. 195–214, ISBN: 978-3-540-49252-8. DOI: 10.1007/3-540-60675-0_46.

[35] J. Svenningsson and E. Axelsson, "Combining deep and shallow embedding for EDSL," in *Trends in Functional Programming*, H.-W. Loidl and R. Peña, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 21–36, ISBN: 978-3-642-40447-4. DOI: 10.1007/978-3-642-40447-4_2.

[36] I. S. Diatchki, M. P. Jones, and R. Leslie, "High-level views on low-level representations," *ACM SIGPLAN Notices*, vol. 40, no. 9, pp. 168–179, 2005-09-12, ISSN: 0362-1340. DOI: 10.1145/1090189.1086387.

[37] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "A programmable programming language," *Communications of the ACM*, vol. 61, no. 3, pp. 62–71, 2018-02-21, ISSN: 0001-0782. DOI: 10.1145/3127323.

[38] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "The Racket manifesto," in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 32, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 113–128, ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.113.

[39] P. J. Landin, "The next 700 programming languages," *Communications of the ACM*, vol. 9, no. 3, pp. 157–166, 1966-03-01, ISSN: 0001-0782. DOI: 10.1145/365230.365257.

[40] D. Leijen and E. Meijer, "Domain specific embedded compilers," *ACM SIGPLAN Notices*, vol. 35, no. 1, pp. 109–122, 2000-12-31, ISSN: 0362-1340. DOI: 10.1145/331963.331977.

[41] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, no. 4, 196–es, 1996-12-01, ISSN: 0360-0300. DOI: 10.1145/242224.242477.

[42] M. Grebe and A. Gill, "Haskino: A remote monad for programming the Arduino," in *Practical Aspects of Declarative Languages*, M. Gavanelli and J. Reppy, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 153–168, ISBN: 978-3-319-28228-2. DOI: 10.1007/978-3-319-28228-2_10.

[43] K. Claessen and D. Sands, "Observable sharing for functional circuit description," in *Advances in Computing Science ASIAN99*, P. S. Thiagarajan and R. Yap, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1999, pp. 62–73, ISBN: 978-3-540-46674-1. DOI: 10.1007/3-540-46674-6_7.

[44] A. Gill, "Type-safe observable sharing in Haskell," in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, ser. Haskell '09, New York, NY, USA: Association for Computing Machinery, 2009-09-03, pp. 117–128, ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596653.

[45] O. Kiselyov, "Implementing explicit and finding implicit sharing in embedded DSLs," *Electronic Proceedings in Theoretical Computer Science*, vol. 66, pp. 210–225, 2011-09-01, version: 1, ISSN: 2075-2180. DOI: 10.4204/EPTCS.66.11. arXiv: 1109.0784.

A

E core definitions

This appendix presents some core parts of the implementation, such as the `E` data type.

The code base for the reference implementation of the pattern matching technique presented in the report can be found at the following GitHub repository: `https://github.com/OctopiChalmers/hpatterns`. The version of Haski that includes the integration of the pattern matching technique is available at the Haski GitHub repository, under a separate branch[1].

A.1 The generic expression language `E`

The full definition of the `E` data type.

```
{-# LANGUAGE GADTs                #-}
{-# LANGUAGE ScopedTypeVariables #-}

module E.Core where

import Data.Bits (Bits)
import Data.Proxy (Proxy(..))

import E.CTypes (CType)


data E a where
    - Representation of pattern matching using a case expression.
    ECase :: (CType a, CType b)
        => Scrut a
        -> [Match p b]
        -> E b
    ESym :: ScrutId -> E a

    - Constructor for tagging expressions corresponding to
```

[1]Latest commit as of this thesis: `https://github.com/OctopiChalmers/haski/tree/d226b24e574fe3534e0afb1bc227b3b08020861c`

```
    - constructor fields.
    EField :: ArgId -> E a -> E a

    - Straightforward operators.
    EVal :: a -> E a
    EVar :: String -> E a

    EAdd :: (Num a) =>          E a -> E a -> E a
    EMul :: (Num a) =>          E a -> E a -> E a
    ESub :: (Num a) =>          E a -> E a -> E a
    EDiv :: (Fractional a) =>   E a -> E a -> E a
    EGt  :: (Num a, CType a) => E a -> E a -> E Bool
    ELt  :: (Num a, CType a) => E a -> E a -> E Bool
    EGte :: (Num a, CType a) => E a -> E a -> E Bool
    ELte :: (Num a, CType a) => E a -> E a -> E Bool
    EEq  :: (Eq a, CType a) =>  E a -> E a -> E Bool
    ENeq :: (Eq a, CType a) =>  E a -> E a -> E Bool
    ENot ::                     E Bool -> E Bool
    EAnd ::                     E Bool -> E Bool -> E Bool
    EOr  ::                     E Bool -> E Bool -> E Bool

    ECFloorInt    :: E Double -> E Int
    ECFloorDouble :: E Double -> E Double

    - Constructors representing bitwise operations.
    ECast   :: (CType a, CType b) => E a -> Proxy b -> E b
    EShiftL :: (Bits a) => E a -> E Int -> E a
    EShiftR :: (Bits a) => E a -> E Int -> E a
    EBitAnd :: (Bits a) => E a -> E a -> E a

{- | A (Match p b) represents one branch in a pattern match, i.e.
one constructor of a sum type p. p is the type pattern matched on,
and b is the return type of the body, i.e. the right hand side of a
case-of arrow.
-}
data Match p b where
  Match :: forall p b . CType b
    => E Bool     - ^ Predicate for selecting this particular branch.
    -> E b        - ^ Body of branch; what is returned if chosen.
    -> Match p b

data Scrut a = Scrut (E a) ScrutId
newtype ScrutId = ScrutId Int deriving (Eq, Ord, Show)
```

A.2 The CType type class

The definition of the CType class and all implemented instances.

```
{-# LANGUAGE AllowAmbiguousTypes #-}

module E.CTypes where

import Data.Char (toLower)
import Data.Int (Int8)
import Data.Word (Word8, Word16)

class CType a where
    ctype   :: String       - Name of type in C
    cval    :: a -> String  - Translation function for values
    cformat :: Char         - Used for printf formatting

instance CType Int where
    ctype = "int"
    cval = show
    cformat = 'd'

- Requires <stdint.h>
instance CType Int8 where
    ctype = "int8_t"
    cval = show
    cformat = 'd'

- Requires <stdint.h>
instance CType Word8 where
    ctype = "uint8_t"
    cval = show
    cformat = 'd'

- Requires <stdint.h>
instance CType Word16 where
    ctype = "uint16_t"
    cval = show
    cformat = 'd'

instance CType Double where
    ctype = "double"
    cval = show
    cformat = 'f'

- Requires <stdbool.h>
instance CType Bool where
    ctype = "bool"
    cval = map toLower . show
    cformat = 'd'
```