Bachelor Thesis

Partial Evaluation in Template Haskell

Klara Schlüter

Supervisor: Prof. Christian Lengauer, Ph.D.

Tutor: Dr. Armin Größlinger

7th January 2019

Abstract

Partial evaluation is a powerful program specialisation technique. For fixed values for some of a program's parameters, all parts of the program that depend only on those fixed parameters are precomputed. The result is a more efficient residual program with the same semantics as the original one. In this work, a prototype of a partial evaluator is successfully implemented. It is capable of specialising functions written in the purely functional programming language Haskell. Metaprogramming functionality is required when manipulating program definitions, e.g. when code for a specialised verison of the original function is generated. In contrast to an earlier approach to partial evaluation for Haskell, this implementation uses the language extension Template Haskell, which provides functionality for metaprogramming in Haskell. This thesis provides test cases for the prototype, revealing the capabilities of partial evaluation: significant improvements of computation time and space are measured for Fibonacci computation. As a real world application, useful for example in image processing, the partial evaluator is applied to matrix multiplication in order to remove double computation. An outlook concerning further implementation features is given, for example a method preventing some more cases from non-termination is explained.

Contents

1 Introduction

When implementing a program, there is often a trade-off between elegance and efficiency: The elegant solution is to implement one program solving numerous similar problems. This solution is very readable and maintainable, since it avoids boilerplate code. It is an **abstract** solution. Another approach is to write many, very specialised and therefore highly efficient programs, which are very **concrete**.

Partial evaluation tries to combine both. Implementing the abstract, readable code, partial evaluation can be used to automatically specialise it to different parameters. Given values for some of the function's parameters, the partial evaluator precomputes as much as possible to generate a specialised, faster program. These concrete programs can then be the ones executed at runtime.

Specialising programs requires the ability to analyse, modify and even generate program code. This is called metaprogramming, as it is a technique working on the structure and properties of a program instead of executing it. Metaprogramming can not easily be used for every programming language. It requires a manipulable representation of code and constructs to manipulate code like program data. For the purely functional programming language Haskell, an easy way of metaprogramming was introduced in 2002: The language extension Template Haskell.

Before Template Haskell was introduced, Silvano Dal-Zilio [1] made a first attempt to implement partial evaluation in Haskell. Without the Template Haskell possibility to manipulate Haskell Code, he had to define a smaller language, Tiny Haskell. His partial evaluator specialises Tiny Haskell programs and is written in Haskell. Two years after the introduction of Template Haskell, Duncan Coutts [2] implemented a library offering functionality that is similar to classic partial evaluation. Instead of generating specialised programs, his program produces generating extentions for given functions. A generating extension is able to specialise exactly one function to a given parameter, while a partial evaluator applys to every given function. Duncan Coutts uses Template Haskell to express the produced generating extensions, which are themselves programs.

In this work, a partial evaluator for Haskell programs is introduced. With Template Haskell, there is no need to reduce Haskell to a subset language. All Haskell constructs can be specialised. Furthermore, Template Haskell provides the possibility of performing the partial evaluation at compile time, so the code executed at runtime can be optimised without the efficiency of optimisation being an important issue. The partial evaluator will produce a specialised version of a given, arbitrary function to a given parameter value.

This work is structured as follows. The first part motivates and introduces partial evaluation and explains its theoretical foundation and its basic concepts. The second part is concerned with Template Haskell. The reader is given an explanation of the idea of Template Haskell and how it is used for partial evaluation in this work. On this fundament, the two earlier approaches to implement partial evaluation in Haskell mentioned above are discussed and compared to

the approach in this work. The last part first describes the implementation of the partial evaluator in detail. After a discussion of the problems that arose while implementing, testcases, speedups and results are presented. The final two sections list features, that remain to implement and conclude this work and achievements.

2 Partial Evaluation

Consider a set of similar problems with a computable solution. When implementing programs to solve those problems, in general, one can choose on a scale between the following two extremes:

- **One general solution for all problems**: One possibility is to abstract from the single problem as far as possible. This means to introduce parameters to model the different aspects of the problems and take the common aspects to design an algorithm to solve the problems. The implementation of this solution is then only one parameterised program that is able to solve an arbitrary problem from the set. A certain problem to solve is specified by fixing the parameters to values.
- **A special solution for every distinct problem in the set**: The other end of the scale is to find for every problem the most efficient solution and implement it. This leads to as many programs as slightly different problems.

For example, we want to write a program in which we will compute the power x^n for some, already known values of n, let's say $n \in \{1, 2, 8, 12, 18, 28, 128, 182\}$, and unknown $x \in \mathbb{N}$. We can implement different solutions:

```
-- Parameterised implementation for all problems:
power x n = if n == 0
               then 1
            else
               x * (power x (n-1))

-- Special implementations for the single problems:
power1 x = x
power2 x = x * x
power128 x = x * ... * x
```

Choosing the parameterised, general solution yields better maintainability and modularity, the code is reused for many problems, which leads to shorter programs which are easier to understand and faster to implement. But, as the power example illustrates, the special programs are often much faster, as there is no unnecessary calls, parameter passing, and the computation is reduced to a minimum [2].

Partial evaluation is a technique trying to combine those two approaches to avoid the tradeoff between efficiency and simplicity. The idea is to precompute

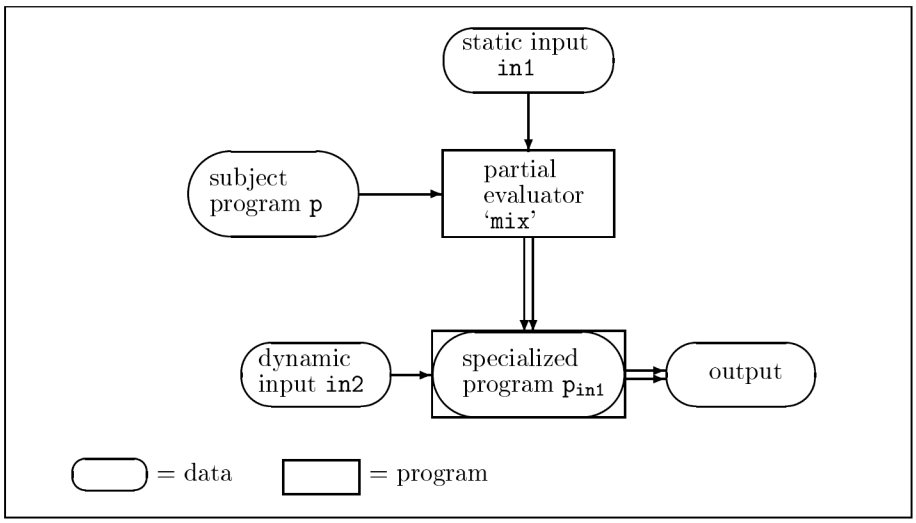

Figure 1: The partial evaluation process [3].

everything that is possible if some of the input values to a program are already known while others are still unknown. A partial evaluator is a program that accepts as arguments a program `p` and some input value `x` for specified parameters of p and then specialises `p` to `x`. The resulting specialised program is called a residual program.

Using a partial evaluator `mix`, we can rewrite the power program:

```
-- General solution
power x n = if n == 0
              then 1
           else
              x * (power x (n-1))

-- Generated efficient solutions:
power1 = mix power 1
power2 = mix power 2
power182 = mix power 182
...
```

Andrei Ershov called partial evaluation **mix**, since it is a mixture of evaluation and program generation [3]. This name was adopted by the major part of literature about partial evaluation. Implementing the general solution, we use `mix` to specialise it to the given n, and obtain one specialised version of the function `power` for every $n \in \{1, 2, 8, 12, 18, 28, 128, 182\}$. While preserving the benefits of the general solution, `power`, because it is the one to implement, understand and maintain, we can execute the efficient special solutions, the residual programs `power1, ..., power182`.

This first section explains the theoretical foundation, characteristics and techniques of partial evaluation. The partial evaluator in this work, see Section 5, is implemented in Haskell, a functional language. The structure to be specialised are functions. Therefore, the word "function" is used equivalently to "program" in the following.

2.1 Theoretical Foundation: Kleene's s-m-n Theorem

The theoretical existance of such a function `mix` is given by Stephen Cole Kleene's *s-m-n theorem*. The theorem is a result of his research in the field of computability. The (slightly simplified) statement of the theorem is the following [4]:

Let $f : N_1 \times ... \times N_k \times M_1 \times ... \times M_l \rightarrow X$ be a computable function. Then for all $(n_1, ...n_k) \in N_1 \times ... \times N_k$ exists a computable function $g_{(n_1,...n_k)} : M_1 \times ... \times M_l \rightarrow X$ with the property:

$$\forall (m_1, ...m_l) \in M_1 \times ... \times M_l : f(n_1, ...n_k, m_1, ...m_k) = g_{(n_1,...n_k)}(m_1, ...m_k) \quad (1)$$

The function $g_{(n_1,...n_k)}$ is computable from f and $n_1, ...n_k$ [2].

A little less technical: For every function and fixed values for some of its parameters, a residual function taking less arguments exists and gives the same result for the same inputs. And not only this: Generating such a residual function for the given original function and its fixed inputs is computable. Specilisation of a function to some of its input values is hence possible.

The sheer existence of the specialised function is quite intuitive: The trivial partial evaluator just constructs a function applying the original function and the fixed parameters. For the orignial function f defined as above and the given parameters $n_1, ...n_k$, the residual function $g_{(n_1,...n_k)}$ would be defined as:

$$g_{(n_1,...n_k)}(m_1, ...m_l) = f(n_1, ...n_k, m_1, ...m_l)$$

The Equation 1 obviously holds for this definition of g.

Kleene used this idea in the first proof of the s-m-n theorem [5]. He gives a Turing-Machine constructing those trivial residual functions [3]. A different proof of the theorem, using Gödel numberings and properties of Gödel numberings, can be found in [4].

These trivial residual functions are easy to compute, but do not provide a faster performance, they are even less efficient than the original functions due to the extra function call. For computability theory as in [5] and [4], the efficiency does not matter. In practice, partial evaluation would not be of any use, if the residual functions were less efficient than the original functions. Still, the power example at the beginning of this chapter shows that at least for some functions and inputs, there is a residual function that is expexted to run faster. Partial evaluation is the science of automatically computing such an efficient residual function, if possible.

2.2 Staged Computations

As Figure 1 visualises, `mix` generates a program that is then run in a next step. These kind of steps are called stages in programming. A multistage program is written for two or more stages. The computations in one stage result in functions, program parts or other data structures, that are used to build the next stage. E.g., stage one generates code which is then executed as the following stage [2].

Computations using partial evaluation take place in at least two stages. In the first stage, a given program is specialised by partial evaluation, which produces a residual program. The residual program is then executed in the second stage. In this approach, partial evaluation will be performed at compiletime to optimise the code executed at runtime. The two stages are hence compiletime and runtime. Variables with values known in a current stage are called **static**. Unknown variables, with values known in the next stage, are called **dynamic**. Programs are specialised to their static arguments.

Here, partial evaluation takes place at compile time. Thus, from now, data known at compile time is called static, data unknown at compile time (unknown until runtime) is called dynamic. For simplicity, `mix` assumes that only the first parameter of the function to specialise is static. The signature of `mix` will thus be something like

```
mix :: f -> s -> r
```

where

```
f :: s -> d1 -> ... -> dn -> o
```

is the function to specialise, with `s` a value for its static parameter, given to `mix` as second argument, and $\texttt{d}_1$ to $\texttt{d}_n$ are dynamic. The returned residual function

```
r :: d1 -> ... -> dn -> o
```

is then a function taking only values for the dynamic arguments and returning the same output as `f`.

Specialising the given function only to its first argument assumes the input function is of a certain format in two aspects, as following. Both do *not* limit the possibilities of partial evaluation.

- **Specialising to only one argument**: To specialise a function to multiple static arguments, `mix` can be applied multiple times, specialising the residual function to one more static argument in each iteration. Applying `mix` multiple times does not require multiple stages, since the resulting code of each iteration is not executed.

- **Specialising to the first argument**: The arguments of a function `f :: d1 -> s -> ... -> dn -> o` can easily be switched, for example using a lambda expression of the form `(\ d1 s ... dn -> f s d1 ... dn)`. Hence, a function can be specialised to any of its arguments by introducing a new function that calls `f` with changed argument order.

2.3 Online and Offline Partial Evaluation

The main principle in partial evaluation is to precompute everything that is computable if a value for the static parameter is given. `mix` decides what expressions will be precomputed based on a classification into static and dynamic. The classification of expressions depends on which parameter is static [6]. It has to be congruent, that means: an expression, that contains a dynamic parameter or another dynamic expression, has to be classified as dynamic [3]. The more expressions are classified as static, the more can be precomputed, the less has to be computed at runtime, the faster is computation at runtime. Hence, an optimal classification classifies as many expressions as static as possible, that is, every expression that depends only on other static expressions or the static parameter.

There is one problem: finding an optimal classification is uncomputable, as a result of the halting problem [7]. One either has to sacrifice optimality for termination, or vice versa. It is always possible to compute just any classification. The trivial classification for example classifies every expression as dynamic, which is always computable. But this usually is far from optimal, as nothing can be precomputed. In contrast, an algorithm looking for an optimal solution has to explore every detail of an expression, which can lead to nontermination. E.g., consider a function

```
f x y = f x (y + 3)
```

To classify an expression `f a b`, an optimal algorithm has to look at the function body in case it somehow assigns `a` and `b` to a static value. In the body, it will find the next call and unfolds it, leading to an endless loop. The process of classifying expressions as static or dynamic is called binding time analysis (BTA).

Partial evaluation now consists of two tasks: BTA and specialisation. It is distinguished between **offline partial evaluation**, where BTA is done before specialisation, and **online partial evaluation**, where both happens at the same time. Offline partial evaluation is often easier to implement and debug [6], and might be more efficient at specialisation time [8]. Online partial evaluation achieves better classifications, since potentially already precomputed expressions can be used for the classification of other expressions. The realisation of online partial evaluation is more complex [3].

2.4 Notation

Another observation in Figure 1 is, that the obtained special program has to be both program data which can be passed around and manipulated by `mix`, and itself executable program code. As data, the program is needed in its semantic representation; a representation, not necessarily in a certain language, that holds the semantics of what the program computes and how. As executable program, it is needed in its syntactic representation in a certain language, so it can be interpreted or compiled and executed. In the following, the syntactic representation of a program `p` in a language l is denoted as $[\![\ p\ ]\!]_l$, and if the

language is not relevant, as $[\![p]\!]$. Its semantic representation is written without a special notation, in the same way as any other program data: p.

With this notation, for the application of a function f to i_s and i_d, we write:

$$[\![f]\!]\ i_s\ i_d = result \tag{2}$$

f is a program, so the syntactic representation is needed to execute it on the data i_s and i_d. When applying `mix`, f is needed in its semantic representation.

$$[\![[\![mix]\!]\ f\ i_s]\!]\ i_d = result \tag{3}$$

where the output of `mix` is a semantic representation until it is converted to an executable representation to be applied to i_d. Combining Equations 2 and 3, we get the property defining mix.

$$\forall f,\ i_s,\ i_d :\ [\![f]\!]_l\ i_s\ i_d = [\![[\![mix]\!]_{l_{mix}}\ f\ i_s]\!]_l\ i_d$$

where f is a function, i_s the static argument it is specialised to, and i_d the dynamic arguments. The statement of this equation is: a function, specialised using `mix`, should yield the same results as the original function.

Two things concerning the language should be noted:

1. The language of the syntactic representation of `mix`, l_{mix}, is not relevant for the process of partial evaluation.

2. `mix` expects as first argument not the syntactic representation of f, but the semantic representation. It also returns a semantic representation of the residual program. Thus, the language we are implementing f is not relevant, too (as long as there is a way to convert to some semantic representation that is understood by `mix`).

2.5 Realisation

Concluding this section, the partial evaluator introduced in Section 5 is classified according to the characteristics described. A partial evaluator for Haskell functions will be presented. It is implemented using Template Haskell[1], which provides a way of converting a program $[|\ f\ |]_l$ to its abstract syntax f. Binding time analysis will be performed online. Partial evaluation takes place at compile time, which is thus the first stage, while the obtained code is executed at runtime, the second stage. The partial evaluator accepts as arguments the function to specialise and a value for its first parameter. It returns a residual version of the function, specialised to the given value as its first parameter.

2.6 The Futamura Projections

The first to use the term *Partial Evaluation* in literature were Lionello L. Lombardi and Bertram Raphael [9] in 1964. Writing about *Incremental Computation*, which is to "evaluate[...] expressions under the control of the available

[1] A short introduction to Template Haskell is given in Section 3.

information context" [9], they conclude their work under the name Partial Evaluation. Only 7 years later, Yoshihiko Futamura [10] introduced three famous equations describing the possibilities of generating a compiler by applying a partial evaluator to itself. They were later called the *Futamura Projections* by Andrei Ershov [3].

Futamura's initial findings include that application of a partial evaluator to an interpreter yields a function that somewhat compiles its input by translating it into the output language of the partial evaluator. The following explanation of this statement is guided by the explanation given in [2].

A program *source* assigns certain inputs *in* to specific outputs *out*.

$$[\![\ source\ ]\!]_{L_{source}}\ in = out$$

Per definition, an interpreter $[\![\ int\ ]\!]_{L_{target}}$ for programs in the language L_{source} computes the corresponding *out* from *source* and a given *in*.

$$[\![\ int\ ]\!]_{L_{target}}\ source\ in = out$$

Since an interpreter is a program, it can be specialised to a value using a partial evaluator. The partial evaluator *mix* must be applicable to programs in the language L_{target} the interpreter is written in.

$$[\![\ [\![\ mix\ ]\!]\ int\ source\ ]\!]_{L_{target}}\ in = out$$

With the substitution $target = [\![\ mix\ ]\!]\ int\ source$ it is clear, that applying the partial evaluator *mix* (returning programs in the language L_{target}) to an interpreter *int* for programs in language L_{source}, yields a program *target* with the property:

$$[\![\ target\ ]\!]_{L_{target}}\ in = out$$

A program *source* in a language L_{source} thus has been compiled to a program *target* in a language L_{target} with equivalent semantics. Compiling a program using an interpreter and a partial evaluator is desirable, since interpreters are easier to implement, to debug and to understand [11].

The second idea is just an application of the definition of partial evaluation: For a program `p`, that, applied to a certain value, does a certain job, partial evaluation can be used to generate a program doing only this job.

The **first Futamura projection** is given above:

$$target = [\![\ mix\ ]\!]\ int\ source$$

A partial evaluator applied to an interpreter compiles its input. Since a partial evaluator is a program itself, partial evaluation can be applied again to generate a compiler, which is represented by the **second Futamura projection**.

$$compiler = [\![\ mix\ ]\!]\ mix\ int$$

Applying a partial evaluator to a program that generates compilers for specific inputs, we obtain the **third Futamura projection**, a program generating compiler generators if applied to an interpreter.

$$cogen = [\![\ mix\ ]\!]\ mix\ mix$$

Although these ideas where theoretically developped quite early, they were not applied as soon. The problem was to implement a partial evaluator that is self-applicable. In 1985, Neil D. Jones, Peter Sestoft and Harald Søndergaard [12] succeeded to implement a self-applicable partial evaluator for a language similar to Lisp and used it to compile, generate compilers and generate compiler generators [3]. Due to the language extension Template Haskell, this approach to a partial evaluator is not self-applicable without further work. In chapter 3.5, the conflict between TemplateHaskell and the Futamura projections is discussed in detail.

3 Template Haskell

Haskell is a strongly typed, purely functional, lazy language. As a functional language, referential transparency is guaranteed. That means, for a given equation, any occurences of the right side in the code can be replaced by the left side and vice versa. Computations do not have side effects and do not depend on any changing values. As a consequence of these properties, functional languages are perfect for partial evaluation. Partial evaluation of imperative constructs, computations depending on anything else than the computation itself are harder to predict. Haskell's type system helps to generate valid residual functions, since the generation must be type safe, too.

To implement a partial evaluator for Haskell functions, the ability of manipulating function code is required. Furthermore, partial evaluation should be performed at compile time to produce runtime code. For these tasks, Template Haskell comes in handy. A short tutorial on the use of Template Haskell is given in this section.

3.1 Overview

Template Haskell is a language extension to Haskell. It was originally proposed by Tim Sheard and Simon Peyton Jones in [13] in 2002. Since ghci x, a slightly adapted version is included in ghci. Template Haskell allows type safe meta programming in Haskell, i.e. type safe code manipulation and generation during compile time.

To do so, Template Haskell implements three features:

1. In addition to their concrete syntax representation (as a Haskell program $[|\ p\ |]_{Haskell}$), Haskell language constructs, e.g. function declarations, can be represented as **abstract syntax trees** (as Haskell program data p) [14]. This is implemented as a library of algebraic data types, described in Section 3.2. Abstract syntax trees can be passed between functions, manipulated and generated.

2. **Quotation brackets** `[|...|]` and **splice operator** `$`, offer the possibility to convert from concrete syntax to the corresponding abstract syntax

tree and back. Besides, the `$` operator initiates the execution of computations during compile time. Section 3.3 summarises the functionality of these two new language constructs.

3. Computations concerning those syntax trees are all wrapped in the **quotation monad Q**. The Q monad is necessary to handle naming issues and holds compile time information, as explained in 3.4.

To use the data types forming the abstract syntax trees, `Language.Haskell.TH` has to be imported. Splice operator, quotation brackets and the quotation monad can be enabled with the flag `-XTemplateHaskell`.

When using Template Haskell for meta programming, Haskell is "both [...] the manipulating language and the language being manipulated" [14]. The data types representing the abstract syntax trees are implemented in Haskell themselves and can be manipulated using Haskell syntax. The Q monad behaves like every monad in Haskell. To manipulate Haskell, we can use Haskell.

As promised, meta programming in Template Haskell is type safe. The type checking of the runtime code happens in three steps:

1. The module's compile time code (all toplevel computation that is contained in a splice) is type checked and compiled. This includes the abstract syntax trees and all splices and quotations in the compile time code.

2. The compile time code is executed. This produces code snippets to be executed at runtime. The snippets are inserted into the module.

3. The resulting runtime code is type checked and compiled together with the user written code.

3.2 Abstract Syntax Trees

The abstract syntax trees can be encoded by recursive data types defined in the library `Language.Haskell.TH`. The constructors of the data types represent kinds of nodes. For every child, the constructor has an argument of the corresponding type. E.g., the constructor `CondE` of type `Exp`, that models an if-then-else construct, has three `Exp` arguments, one for the if case, one for the else case, and one for the condition..

```
Exp = ... | CondE Exp Exp Exp | ...
```

In this work, the most important data types are:

- `Exp` for expressions, e.g. `2 + x`,
- `Dec` for declarations, e.g. `id :: a -> a, id a = a`,
- `Pat` for pattern matching, e.g. the `(x:xs)` in `(\(x:xs) -> xs)`,
- and `Name` for variable names.

The full documentation can be found at [15]. With all data types in the library, every language construct in Haskell can be encoded in abstract syntax. As an example, let foo be a function with the following concrete syntax representation:

```
foo :: Integer -> Integer
foo 0 = 21
foo x = 8
```

The corresponding abstract syntax is given by:

```
[ SigD foo (AppT (AppT ArrowT (ConT Integer)) (ConT Integer))

FunD foo [
    Clause [LitP (IntegerL 0)] (NormalB (LitE (IntegerL 21))) []
    Clause [VarP x] (NormalB (LitE (IntegerL 8 ))) []
   ]
]
```

Furthermore, the library contains some auxiliary functions for the Q monad, constructors lifted to Q, and a pretty printer to output the verbose syntax trees in a readable, more concrete syntax like style.

3.3 Quotes and Splices

The quotation brackets `[|` and `|]` convert the concrete syntax inside the brackets to an abstract syntax tree as a computation in Q. There are four kinds of quotation brackets [16]:

- `[e| x |]`, abbreviated `[| x |]`, is of type Q Exp, where `x` is an expression in concrete syntax,
- `[d| x |] :: Q Dec` with `x` a declaration,
- `[t| x |] :: Q Type` with `x` a type,
- `[p| x |] :: Q Pat` with `x` a pattern.

To "quote" a variable name, an apostrophe `'` can be used: `'x :: Name` is the `Name` of the variable `x` that is currently in scope. The `Name` contains for example module information, to distinguish from other variables named `x` in different scopes.

It should be noted that the construct between the brackets is interpreted syntactically (as the concrete syntax of a program part), not semantically. That means, expressions between brackets are not evaluated. As an example, `[| 1 + 0 |]` evaluates to

```
(Q) InfixE (Just (LitE (IntegerL 1)))
           (+)
           (Just (LitE (IntegerL 0))),
```

not (Q) `LitE (IntegerL 1)`.

Unfortunately, the quotation brackets perform exactly the opposite convertion to the brackets introduced in Section 2.4 [2]. Nevertheless, since the literature about partial evaluation concordantly uses the brackets to convert from abstract to concrete syntax, both semantics of the brackets are used in this work. In code pieces, the brackets refer to the Template Haskell quotation brackets, and are written in typewriter font: `[| |]`. In theoretical ideas, the brackets refer to the brackets introduced in Section 2.4, and are written in equational mode, $[\![\ ]\!]$, as in Section 2.4.

The splice operator `$`, the inverse of the quotation brackets, converts back to concrete syntax. For example:

$$\texttt{\$([| 3 |])} \rightsquigarrow \texttt{3}$$

Given a function `f x = [| x |]`:

$$\texttt{\$(f 3)} \rightsquigarrow \texttt{3}$$

The construct in the parentheses has to be a computation in Q. Wherever the splice operator appears in a module, the enclosed monadic Q computation is executed during compile time. The result is converted to concrete syntax and replaces the splice before the module is compiled as usually.

If the construct to splice in is represented by a variable, the parentheses can be omitted: `$x` splices in the monadic Q computation represented by the variable `x`. Written with no space between the `$` and the parentheses or name, the splice operator overwrites the meaning of the `$` character that replaces parentheses.

Template Haskell is designed for programing in two stages: compile time and runtime. As a result, it is not possible to splice a construct in the module where it is defined. To do this, multiple compilation stages are required [14], as the content of the splice needs to be resolved before the splice is resolved to concrete syntax.

3.4 The Q Monad

The quotation monad Q offers two features, that are important for this work.

Generation of fresh names: In case code is generated in one scope and later spliced in in another scope, the Q monad keeps track of names visible in the different scopes. Using the function `newName :: String -> Q Name`, it is possible to ask the monad to generate a fresh name that will not be captured in any splice (i.e., that will not be identified with any other names visible in the scope of the splice). Still, if desired, it is possible to generate capturable names using the function `mkName :: String -> Name` or even lookup the name of a variable currently in scope (see Section 3.3).

Reification: As stated before, the monad holds compile time information. This information can be queried using the function `reify :: Name -> Q Info`. The returned `Q Info` is structured differently depending on value corresponding to the given name. It contains for example type and implementation of the value.

The monad supports the usual operations return, bind, and fail, and can be used with the do-notation. The library `Language.Haskell.TH` provides the function `runQ` and `runIO` to convert computations in the IO monad to computations in the Q monad and vice versa.

3.5 Template Haskell conflicts with the Futamura Projections

Template Haskell does not provide constructs to represent splice operator and quotation brackets in an abstract syntax tree. Hence, programs using Template Haskell constructs can neither be manipulated nor generated. For example `[| [| 4 |] |]` will produce an error. Every occurence of the splice operator or quotation brackets at the top level of a module is resolved during compile time. (Occurences at the toplevel are at the outermost level of a declaration. Occurences inside code manipulating functions are not toplevel.) The code compiled to be executed at runtime usually does not contain splices or quotation brackets anymore.

In consequence, this implementation of a partial evaluator using Template Haskell can not be used in the second and third Futamura projection. In Section 2.6, the second and third Futamura projection were introduced as follows.

Second Futamura projection:

$$compiler = [\![\ mix\ ]\!]\ mix\ int$$

Third Futamura projection:

$$cogen = [\![\ mix\ ]\!]\ mix\ mix$$

In these equations, the partial evaluator *mix* is used to specialise itself to some static argument. This would require the function `mix` itself to be represented in the abstract syntax that it accepts as argument, which is not possible in Template Haskell.

A way of circumventing this problem is shown by Duncan Coutts in [2]. He implements his own functions for converting from concrete Haskell syntax to abstract Template Haskell syntax, defining a data type representing splice operator and quotation brackets. This function introduces one more encoding layer. It has to be used as innermost layer, that is, the layer that is resolved firstly. The second and last conversion to concrete syntax has to produce regular runtime code.

4 Earlier Approaches

Even though there are numerous different implementations of partial evaluators, only two approaches were implemented for Haskell or in Haskell until now. In this chapter, both are shortly presented and compared to the implementation proposed in Section 5.

4.1 1993

The first attempt to implement a partial evaluator in Haskell was made by Silvano Dal-Zilio [1]. He implemented the partial evaluator in 1993, before the introduction of Template Haskell in 2002, which brings two disadvantages. Firstly, without the Template Haskell quotation, the possibility to convert the Haskell code $[\![\ p \ ]\!]_{Haskell}$ to a manipulable structure p, Dal-Zilio needed his own structure to represent Haskell code. He designed Tiny Haskell, in which a subset of all Haskell constructions can be represented. It is, like Template Haskell, implemented in Haskell as data types and therefore can be manipulated using Haskell. The partial evaluator accepts as input and produces as output programs in Tiny Haskell, while it is written itself in Haskell. Secondly, there was no possibility of running code at compile time, so partial evaluation had to take place at runtime.

Executing partial evaluation at runtime meant to take efficiency of the process into account. Maybe this is one of the reasons that offline partial evaluation was implemented in [1]. The specialisation thus happens after binding time analysis according to annotations. Tiny Haskell provides the structure for annotations, showing which expressions are static and which calls are recursive calls. Recursive calls aren't unfolded to avoid nontermination caused by infinite unfolding. Instead, a residual call to a specialised function is generated. Dal-Zilio does not implement binding time analysis. The annotations were expected to be added by the user of the partial evaluator. This assumption bypasses the problem of computing an optimal classification in static and dynamic, but forces the user to annotate his programs.

As the name of the paper, *A self-applicable partial evaluator for a subset of Haskell*, says, Dal-Zilio was aiming for self-application of his partial evaluator as proposed in the Futamura projections.

The three main differences of this work compared to the approach in of Silvano Dal-Zilio are:

- Template Haskell is used. With quotation, we are not limited to a small subset of Haskell, and Haskell code can easily be converted to its abstract syntax. Furthermore, we perform the partial evaluation at compile time, so the runtime code is optimised.
- Our implementation uses online partial evaluation and therefore potentially results in more efficient residual functions. The specialisation time might be longer, but since it happens at compile time, this is less important.
- In this work, the binding time analysis is done automatically while specialising. An advantage of this is better usability: the user does not need any knowledge about how partial evaluation is done.

This work is hence distinct in several points from what was done in [1].

4.2 2004

In 2004, Duncan Coutts [2] implemented a second partial evaluator in Haskell. He describes a prototype for the partial evaluator he plans on building and refers to a thesis to write. Unfortunately, the actual thesis could not be found, so this section is just about the properties of the prototype. As the focus of his work, Duncan Coutts specifies the developement of a partial evaluator applicable to real world problems, instead of theoretical work about partial evaluation. He thus implements a library providing partial evaluation for usual Haskell and uses Template Haskell for tasks like getting an abstract syntax and manipulating code.

As in [1], the user has to input binding time annotated code and the specialisation algorithm works offline. Expecting already annotated code avoids the challenges of binding time analysis (e.g. nontermination, see 2.3), but forces the user to deal with the annotatable data structure and to know about the classification used for specialisation. An online strategy is then redundant, as no binding time analysis has to be done. The classification given by the user is not checked in the prototype, eventual errors result in defective output. Duncan Coutts planned to implement a checking process by type inferrence in the main thesis [2].

Instead of implementing the function `mix`, Duncan Coutts applies the cogen approach: His function `genex` accepts as argument only the function to specialise. With this information, he computes a so called generating extension of the function. The generating extension of a function `f` takes as input only the value of its static arguments and outputs `f` specialised to these values. This approach is inspired by the third Futamura projection:

$$cogen = [\![\ mix\ ]\!]\ mix\ mix$$

Looking at this equation, it is clear, that everything, that can be achieved by genex can also be achieved by a self-applicable implementation of mix. Vice versa, applying the static values to the generating extention obtained by genex yields the same result as mix:

$$\forall f,\ s,\ d\ \ [\![\ [\![\ cogen\ f\ ]\!]\ s\ ]\!]\ d = [\![\ mix\ f\ s\ ]\!]\ d$$

where f is the function to specialise, for simplicity reasons taking only one static and one dynamic argument, s its static argument and d its dynamic argument.

Compared to the mix approach as in this work, the use of the cogen approach is expected to be more efficient at compile time, if one function should be specialised several times to different values. By producing one generating extension and then applying the different values to it, redundant computation is minimised. If a function f will only be specialised to one combination of static arguments, the cogen approach is expected to be less efficient then the mix approach, due to the overhead of producing a generating extension instead of computing the output directly and the extra function call when using it.

The cogen approach avoids the difficulties that occur when trying to force evaluation. For this work, the challenge is discussed in Section 6.3. `mix` is like

a partial interpreter, it computes everything that is already computable. `genex` is similar to a compiler. Expressions are transformed, but not evaluated, which is why the problem of forcing evaluation does not appear.

A disadvantage of the cogen approach compared to the mix approach is, that the output of `genex` is another metafunction that manipulates code. The generating extension therefore is produced in abstract syntax, while itself contains Template Haskell constructs to generate the specialised function. As seen in Section 3.5, Template Haskell does not provide an encoding for quotes and splices. As a consequence, the generating extension can not be expressed in the predefined abstract syntax. To solve that, Duncan Coutts had to build a next layer by defining a way of expressing quotes and splices and an own encoding function.

5 Implementation

In this section, a partial evaluator implementation for Haskell functions is described. The implementation makes use of Template Haskell[2], specialising a function to its first argument. Binding time analysis is performed online[3]. The partial evaluation is performed at compile time[4]. The obtained residual functions can be integrated in the runtime code and compiled and executed as user written code.

Specialisation consists of three tasks: [3]

1. Unfolding function calls
2. Precomputing everything that depends only on static parameters
3. Compressing dynamic structures.

For example, given a function `f a b c = (a + c) - (b * c)`, consider an expression `e`.

```
e = (f 4 y 1)
```

Task 1, unfolding function call, includes two subtasks: The application of the arguments `4`, `y` and `1` to the function `f` can be replaced by the body expression of `f`.

```
e = (a + b) - (b * c)
```

To adapt the variables in the body expression to the parameters, we need a mapping, obtained while unfolding: the parameter names of `f` are mapped to the arguments applied to `f`. In this case, the mapping would be something like `[(a,4), (b,y), (c,1)]`.

Performing task 2, we traverse the function body. We can replace every static parameter by its corresponding expression according to the mapping and then compute every static expression.

[2] An introduction to Template Haskell can be found in Section 3.

[3] Section 2.3 provides an explanation and comparison of online and offline partial evaluation.

[4] Partial evaluation uses staged computation, see Section 2.2.

```
e = 5 - (y * 1)
```

The third task, compressing results, means optimizing rules for code, like $x * 1 = x \ \forall x$. Since this implementation performs specialisation at compile time, the residual code is compiled as directly written code. Optimizing is thus a task we can leave to the compiler. In our example, compiling yields the final code.

```
e = 5 - y
```

In functional programming, the structure to specialise by partial evaluation is a function. In Haskell, there are two structures directly representing a function:

- Lambda functions, e.g. `(\x -> x)`,
- function declarations referred to by the function's name, e.g. `f`, given the declaration `f x = x`.

Furthermore, there are some expressions that can represent a function indirectly, e.g. a let expression: `let f x = x in f`. It can just as well represent expressions that are not a function: `let f = 4 in f`. In this implementation, indirectly represented functions are converted to a structure that can be specialised like one of the direct representations. The structure of a lambda function is less complex than the structure of a function declaration. That is why the code for handling lambdas is partially shown in the following section, while the procedure for function declarations is explained without code extracts.

This section is structured as follows: after a subsection about the general structure of the implementation (5.1), the code for the first task is discussed in Section 5.2 and the code for the second task in Section 5.3. The last subsection, Section 5.4, explains the entry point to the partial evaluator, the implementation of the function `mix`, administrating and initiating the specialisation of the given function.

5.1 Structure

As explained in Section 3, the function to specialise is represented as an abstract syntax tree. There are a few data types modelling nodes in this tree. For every data type X modelling nodes that are to specialise, a function *pevalX* is defined. It takes care of specialisation, calling itself recursively for recursive constructors (children in the tree) and calling other functions to specialise children of other data types. For example, for partial evaluation of a `Body` build by the constructor `NormalB`, the function `pevalBody` is used. For the specialisation of its `Exp` component, it contains a call to `pevalExp`.

```
Body = NormalB Exp | GuardedB [(Guard, Exp)]

pevalBody (NormalB exp) ... = ... pevalExp exp ...
```

A function *pevalX* has as first argument a construct of type X, the construct to specialise. A construct of type X can only be replaced by a residual construct of the same type, so the returned residual construct has to be of type X, too.

```
pevalX :: X -> ... -> [(Name, Dec)] -> (Bool, X)
```

In this implementation, binding time analysis is done online, which means that wether a construcct is static or dynamic is devided during specialisation. This information has to be available for recursive callers to build their specialisation and analysis on. To pass the binding time information of nodes between the functions performing partial evaluation on the nodes, every function *pevalX* does not only return a specialised X, but a pair of the residual construct and a boolean flag showing if it is static or dynamic.

To unfold function calls, the declaration of a function is needed. This information can not be obtained by examining the call, containing only the function's `Name`. More about this problem can be found in Section 6.1. As a consequence, every function in our partial evaluator has to pass a list of function declarations corresponding to function names as third parameter.

The three dots in the general structure of *pevalX* represent one further argument, which is different for different classes of functions. For functions belonging to the process of unfolding, it contains the parameter values of the function call. For functions with the main focus of precomputing constructs, it is a mapping of static variables to the expressions they can be replaced with.

5.2 Unfolding Function Calls

Two structures in Haskell directly represent a function: a lambda expression and a function declaration. The core of both is composed of patterns and corresponding bodies. Since this composition is simpler in a lambda than in a function declaration, let us look at a lambda expression in its concrete syntax as an expample.

```
(\ a b c -> if a == 0 then b + c else b / a)
```

At the left side of the arrow `->`, the three variables `a`, `b` and `c` make a pattern. If the lambda is applied to, e.g., `1`, `2` and `3` , the pattern tells us, that every a in the body equals 1, every b equals 2 and every c equals 3.

```
(\ a b c -> ...) 1 2 3
```

At the right side of the arrow, the function body can be found. It defines what is to be computed from the input values.

Unfolding a function call for partial evaluation follows this structure and so does this section. From a pattern and the parameters, a mapping is computed: which variable can be replaced by which static or dynamic expression? The functions implemented for this "partial pattern matching" are presented in Section 5.2.2. Given this mapping, the body structure of a function can be reduced and partial evaluation can be performed on its components, which is explained in Section 5.2.3. But first, the call to `unfold` is examined.

5.2.1 The Call to `unfold`

The expression we want to unfold is a function application, e.g. `f a b`. The abstract syntax tree of this expression is of type `Exp`.

```
AppE (AppE f a) b)
```

`f`, `a` and `b` all are of type `Exp`. `f` is an expression that represents a function. `a` and `b` represent the expressions the function is applied to. They can be static or dynamic.

Since the constructor `AppE` and therefore the whole construct of function application is of type `Exp`, a function call to specialise is found by the function `pevalExp` while traversing an abstract syntax tree. The aim is to call a function `unfold` that takes care of unfolding. A construct of type `Exp` can only be replaced by a residual construct that has type `Exp`, too, so `unfold` has to return an `Exp`. Following the structure introduced in Section 5.1, a residual expression is always returned in a pair combined with a flag showing if it is static or dynamic. As usual[5] the third parameter to unfold is the list of function implementations.

```
unfold :: ... -> [(Name, Dec)] -> (Bool, Exp)
```

As there is one `AppE` constructor per argument, a function application for a function with several arguments has nested `AppE`s. `pevalExp` uses a helper function to recursively unwind the `AppE` constructors. Every step yields one argument, an `Exp`, which is collected in a list. To call unfold, the arguments need to be specialised by a recursive call to `pevalExp`. Since `pevalExp` returns the usual pair of residual expression and static flag, the argument list is of type `[(Bool, Exp)]`. It is given to unfold as third parameter. Unfold is called only if at least one of the collected arguments is static. Otherwise, the whole expression is just reconstructed. This restriction avoids nontermination[6].

```
unfold :: ... -> [(Bool, Exp)] -> [(Name, Dec)] -> (Bool, Exp)
```

The helper function reaches its base case, if the first argument to an `AppE` is something else than another `AppE` expression (see the example at the beginning of this section). This innermost argument is the `Exp` the collected parameters are applied to. Three cases are distinguished:

1. The innermost expression directly represents a function. That is the case for lambda expressions and variables refering to functions.

2. The innermost expression does not represent a function, but is valid to be applied to values. An example is a constructor. It can be applied to components for construction, even if it is not a function that can be unfolded.

3. The innermost expression is of a form that requires partial evaluation of itself. Consider the expression `let f a = 3 in f`. It obviously represents

[5] Why this is necessary is discussed in Section 6.1.
[6] See Section 6.2 for more information.

a function, but as a syntax tree, the root is a let expression, so it has to be further evaluated to realise it.

4. The innermost expression is an expression no values can be applied to, e.g. a literal: `(4, 4) 4` is undefined in Haskell.

In case 2, the `AppE` constructors are simply reconstructed, as no partial evaluation can be done. In case 3, the expression has to be transformed to eventually fit in case one. Expressions of case 3, for which this transformation is not implemented, are treated as case 2. In case 4, an exception is thrown, since the given abstract syntax does not represent a valid Haskell construct. In case 1, unfold is called with the innermost expression as first element. Now, its signature can be completed.

```
unfold :: Exp -> [(Bool, Exp)] -> [(Name, Dec)] -> (Bool, Exp)
```

5.2.2 Partial Patternmatching

The abstract syntax representation of a pattern is a list containing elements of type `Pat`. Every element represents a single pattern for one of the parameters of a function. For example, the pattern "`\_ 3 (a,b)`" in

```
f _ 3 (a,b) = 0
```

is represented by

```
[WildP, LitP (IntegerL 3), TupP [VarP a, VarP b]].
```

When specialising a function call, two questions have to be answered:

1. Does the pattern match the static parameter values?
2. Which pattern in the function body can be replaced by which static expression?

In the example above, if `f` is called with the static value `(2,5)` for the third parameter and dynamic values for the first and second parameter, the answers should be:

1. Yes, the pattern matches.
2. In the body, replace `a` by `2` and `b` by `5`.

Static parameters

The function `staticMatch` answers these questions for one static parameter and the corresponding element of the pattern list.

```
staticMatch :: Pat -> Exp -> Maybe [(Name, Exp)]
```

The answer to the first question is given by the `Maybe` monad in the result type. If the pattern does not match, `Nothing` is returned. If it matches, the result is `Just m`, where `m` is the answer to the second question: a mapping of variable names to expressions.

`staticMatch` is realised recursively. There are three base cases:

- A simple variable pattern, e.g. `VarP x`, corresponds to `x` in `(\x -> 0)`,
- a literal, e.g. `LitP (IntegerL 4)`, corresponds to `4` in `(\4 -> 0)`, or
- a wildcard, e.g. `WildP` corresponds to `_` in `(\_ -> 0)`.

The implementation of these base cases is easy:

```
staticMatch (LitP lp) (LitE le)
       | lp == le  = Just []
       | otherwise  = Nothing
staticMatch (LitP lp) _  = Nothing

staticMatch (VarP v) exp = Just [(v, exp)]

staticMatch (WildP) _  = Just []
```

A literal pattern only matches an expression representing the exact same literal, everything else yields `Nothing`. A variable and a wildcard match every given expression, so no further checks have to be made. For a matching literal and a wildcard, nothing in the body has to be exchanged, hence the mapper is empty. If the pattern is a single variable, it is obvious that it has to be exchanged by the given expression in the body, so the name and the expression make a pair in the mapper.

All other constructors of the data type `Pat` recursively contain one or more values of type `Pat`, so `staticMatch` is called recursively for all contained. Let's look at the implementation for a pattern representing a tuple, for an example:

```
staticMatch (TupP ps) (TupE es) = foldr1 combine recursives
       where  combine = unionMaybe (++)
              recursives = zipWith staticMatch ps es
staticMatch (TupP _) _  = Nothing
```

Both the constructors for a tuple pattern and for a tuple expression take a list of `Pat` respectively `Exp` as the components of the tuple. These components have to be matched recursively, which is done by calling the predefined `zipWith` combinator on the lists. The length of the lists is assumed equal, since function application is assumed not to contain type errors, as tuples of different lenghts would be. The helper function `unionMaybe`[7] combines the results, concatenating the mappers if there are no `Nothing` values.

[7] The implementation can be found in Appendix A.

Dynamic parameters

Patterns corresponding to dynamic arguments can not be directly replaced in the function body. Instead, they are renamed. For example, let `f` be a function.

```
f x (y,z) = y
```

The expression we want to specialise contains a call to this function with a dynamic second parameter `d`.

```
specialisedFunction d = f 3 d
```

When specialising the call, we replace it by its body.

```
specialisedFunction d = y
```

Now we need to rename to preserve the correctness. We will do this with a `let` expression.

```
specialisedFunction d = let (y,z) = d in y
```

The `let` expression contains renaming declarations (like `y = d`) for every dynamic parameter. The declarations are built with the constructor `ValD` of type `Dec`. The helper function `valDec`[8] simplifies construction.

The prefix "static" of the function `staticMatch` already implies that a function `dynamicMatch` could be added in the future. Instead of renaming the whole expression, static parts of a dynamic expression could be identified and added to the replacement mapping produced by `staticMatch` for the static parameters. This leads to more specialisation in some cases, see Section 8 for further discussion. The administration function for the whole pattern, introduced in the following section, is designed to be adapted to additional functionality for dynamic parameters.

The function `patternMatch`

Now, all that is left to do is to assign parameter values to the corresponding pattern, call `staticMatch` or respectively `valDec` and collect the results for all parameters. This overall functionality is integrated in the function `patternMatch`.

```
patternMatch :: [Pat] -> [(Bool, Exp)]
                      -> Maybe ([(Name, Exp)], [Dec])
patternMatch (p:pats) (a:args)  =
              unionMaybe tupConcat processedHead processedTail
       where processedHead = processArg p a
             processedTail = patternMatch pats args
patternMatch []  [] = Just ([],[])
patternMatch _  _ = Nothing
```

[8] The implementation can be found in Appendix A.

```
processArg :: Pat -> (Bool, Exp) -> Maybe ([(Name, Exp)], [Dec])
processArg pat (static, arg) =
       if static
              then case staticMatch pat arg of
                     Just mapper -> Just (mapper, [])
                     Nothing -> Nothing
       else
              Just ([], [valDec pat arg])
```

The arguments are

- the whole pattern (a list of single parameter patterns) and
- a list of pairs for the arguments: the first component is a flag indicating if the argument is static and the second component is an expression representing the value.

The result is again wrapped in the `Maybe` monad, showing if the whole pattern matches. If yes, a pair of the computed replacement mapping and the list of renaming declarations is returned[9].

If a function `dynamicMatch` as proposed in Section 8 is implemented, it can be called in the `else` case of `processArg`. The return type of the function already allows to return a replacement mapping in addition to the declarations.

5.2.3 Constructing the residual Expression

At this point, we know how and when `unfold` is called and its signature. In the introduction of this section, it was stated that all constructs representing functions in its core functionality can be seen as pairs of patterns and bodies. How patterns are processed is explained in the previous section. The partial pattern matching yields a mapping of to replace static variables and a list of declarations to rename dynamic variables. What is missing now is the connection: how is `patternMatch` called? How is the obtained information used to initiate partial evaluation of the body expressions? How do we reduce the function structure to a residual construct replacing the call?

In a lambda, the composition of pattern and body is quite plain, as the syntax shows.

```
data Exp = LamE [Pat] Exp | ...
```

In fact, its only components are one pattern list and one body expression of type `Exp`. `unfold` is thus very simple.

```
unfold (LamE pats exp) args functions = case patternMatch pats args of
       Nothing -> error "Non-exhaustive␣patterns␣in␣lambda!"
       Just (mapper, renamer)
```

[9]The implementation of helper functions `tupConcat` and `unionMaybe`, used for combining results, can be found in Appendix A.

```
        -> let (statE, resE) = pevalExp e mapper functions
               residual = if statE
                             then resE
                          else
                             compress (LetE renamer resE)
           in (statE, residual)
```

If partial pattern matching returns `Nothing`, the only pattern does not match, so the function application is incorrect. The error message is the same as GHC would return in this case. Else, a mapper and a renamer is returned. With the mapper, evaluation of the body expression is initiated. If the residual expression is dynamic, the renaming declarations are applied with a let expression. If it is static, the result can just be returned, since it can not contain the dynamic arguments anymore. The helper function `compress`[10] removes empty renaming declarations for simplicity.

The structure of declarations is more complicated. One function declaration can contain several `Clause`s with each a pattern list and a body expression of type `Body`.

```
data Dec = FunD Name [Clause] | ...

data Clause = Clause [Pat] Body [Dec]
```

For example, a function `f` can define different bodies for an argument equal to `0` and other arguments.

```
f 0 = 1
f n = f (n - 1)
```

Following the structure described in Section 5.1, there is a function responsible for dealing with the structure of a `Clause`.

```
pevalClause :: Clause -> [(Bool, Exp)] -> [(Name, Dec)]
                          -> Maybe (Bool, Clause)
```

It works similar to unfolding a lambda: `patternMatch` is called and with the resulting mapper, the Body is evaluated using `pevalBody`. There are two differences:

1. `pevalClause` returns a `Maybe` result. Because the evaluated clause potentially is not the only one, it is not necessarily the user's fault, if the pattern does not match. Another matching clause could exist in the same declaration, so here, it is returned `Nothing` to inform the caller without throwing an exception.

2. Instead of renaming the residual body to return a residual construct of type `Exp`, a `Clause` has to be constructed, with parameters for the dynamic arguments.

[10] The implementation of helper functions can be found in Appendix A.

A residual `Clause` is considered static, if the contained body is static.

Managing the clauses is the responsibility of the function `pevalDec`. It is the only function called `pevalX` without returning a residual construct of type X.

```
pevalDec :: Dec -> [(Bool, Exp)] -> [(Name, Dec)] -> (Bool, Exp)
```

This is due to the call that is unfolded. The original `Dec` is not contained in the code, but called by its `Name` as a variable. Variables are of type `Exp`. Performing partial evaluation, we try to replace every construct by a specialised construct. As the function call is of type `Exp`, the residual construct we compute from the `Dec` has to be an `Exp`, too.

`pevalDec` calls `pevalClause` for every `Clause` and keeps only those with a result not `Nothing`. Depending on what `Clause`s remain, different measures are taken.

- **No clauses remaining**: The applied arguments do not match any clause. Again, the GHC error message is used: "Non-exhaustive patterns in function".
- **One static clause remains**: The clause is unwrapped to an `Exp` and returned.
- **One dynamic clause remains**: The clause is unwrapped to an `Exp`. Prepending a `let` to the expression, the remaining arguments of the clauses pattern are renamed to the dynamic arguments of the declaration.
- **Several clauses remain**: The declaration is reconstructed and given a new name. Using a let expression, the residual declaration and its call are returned as an `Exp`: `let` residual declaration `in` new name.

What remains to do for `unfold`? `unfold` is given the declaration as a call by its `Name`: `VarE name`. The name is looked up in the given list of function declarations. If a declaration is found, unfolding is delegated to `pevalDec`. If no declaration can be found, nothing can be done but reconstructing the call: the given arguments are applied with the `AppE` constructor. The expression is returned with the flag showing it is dynamic.

5.3 Precomputing static Expressions

The body of a function defines what is to be computed from the parameters. Hence, to anticipate parts of this computations, the function body has to be processed. As already seen, the body of a lambda function simply consists of one expression of type `Exp`.

```
Exp = ... | LamE [Pat] Exp | ...
```

Following the implementation structure introduced in Section 5.1, the function `pevalExp` takes care of such body expressions of type `Exp`.

The body expression of a function declaration is of type `Body`.

```
data Body = NormalB Exp | GuardedB [(Guard, Exp)]
```

Every `Body` indirectly contains a body expression of type `Exp`. The core functionality of evaluating expressions, `pevalExp`, can hence be used. Still we have to handle the possibility of guards by implementing the functions `pevalBody` and `pevalGuard`.

5.3.1 Plain Function Bodies: `pevalExp`

The function `pevalExp` has three parameters: the expression to specialise, the mapping assigning variable names to static expressions, and the list of functions as usual.

```
pevalExp :: Exp -> [(Name, Exp)] -> [(Name, Dec)] -> (Bool, Exp)
```

Every `Exp` is itself a syntax tree. The different constructors represent different kinds of nodes. For partial evaluation three things have to be done:

- Variables, that correspond to static parameters, have to be replaced by the static value. This is done using the given mapping.
- Precomputable expressions have to be identified, that is, binding time analysis has to be done.
- Identified static expressions have to be evaluated.

Like compressing, evaluating can in most cases be left to the compiler.

Since variables are leaves in the syntax tree, and `pevalExp` traverses the tree recursively, the first task is solved by implementing `pevalExp` for the constructor `VarE`. If the variable is contained in the mapper, it is replaced and the expression is returned as static. Otherwise, it is dynamic and can not be replaced.

```
pevalExp (VarE x) mapper functions = case lookup x mapper of
       Just exp -> (True, exp)
       Nothing  -> (False, VarE x)
```

The second task, binding time analyis for every node, is done on the basis of the result of the recursive calls for the node's children. A residual node is build from the residual children. Depending on the binding time information of the children, it is decided if the residual node ist static or dynamic.

As an example, let us have a look at the implementation of `pevalExp` for the recursive constructor `TupE` (representing a tuple).

```
pevalExp (TupE es) mapper functions =
       let residual = TupE (residuals resEs)
           static = allStatic resEs
           resEs = map recursiveCall es
           recursiveCall e = pevalExp e mapper functions
       in (static, residual)
```

A tuple is only static, if all of its components are static. To build a residual expression, its components are simply replaced. The helper functions `allStatic` and `residuals`[11] combine the binding time information or respectively the residual expressions of the components.

5.3.2 Complex Function Bodies

There are two kinds of `Body`s.

```
data Body = NormalB Exp | GuardedB [(Guard, Exp)]
```

A body can be constructed by `NormalB`, containing an `Exp`. In that case, specialisation consists only of calling `pevalExp`. A `Body` is static, if the contained `Exp` is static. The residual `Exp` is wrapped in the constructor `NormalB` and returned.

A body containing guards is represented by the constructor `GuardedB`. In concrete syntax, a guarded body is composed by pairs of a boolean condition and a body expression that is returned if the condition evaluates to `True`.

```
f a b  | a == b  = 4
       | otherwise = 0
```

It is represented by a list of pairs containing a `Guard` and a simple `Exp`. For the guards, the function `pevalGuard` performs binding time analysis and specialisation. Due to the force-evaluation problem, described in Section 6.3, even for a static `Guard`, `pevalGuard` is not able to decide if the condition evaluates to `True`, hence, `pevalBody` calls `pevalGuard` and `pevalExp` for the pairs and reconstructs a guarded body from the residual expressions. Since `pevalBody` has to keep all pairs, a guarded body is only static, if all bodies and all guards are static.

5.4 User Interface

In the previous sections, the functionality of partial evaluation has been implemented. What is left to do is to define an user interface, a function, that initiates partial evaluation.

As seen in Section 2.2, our partial evaluator accepts as arguments the function to specialise, `f`, and a value, `x`. `f` is specialised to `x` as its first argument. Both `f` and `x` are given in their semantic, manipulable representation, as the computed residual function. In Template Haskell, that is an abstract syntax tree wrapped in the quotation monad. It is assumed, that both `f` and `x` represent valid Haskell constructs, and that `x` really is static.

5.4.1 The Function `qmix`

The entry function to this partial evaluator, the interface to the user, is called `qmix`. Its signature is given by

[11] The implementation of helper functions can be found in Appendix A.

```
qmix :: Q Exp -> Q Exp -> Q [(Name, Dec)] -> Q Exp
```

where the first argument represents `f`, the second argument represents `x`, and the third argument is the list of functions[12]. The following example call specialises the function `power`, defined in Section 2, and binds the residual function to the name `power182`. `power182` can be used as any other function.

```
power182 x = $( qmix [| power |] [| 182 |] functions ) x

y = power182 2
```

The implementation of `qmix` is simple. Its only responsibility is unwrapping the quotation monad. With the plain syntax trees, it calls `mix` for partial evaluation.

```
qmix :: Q Exp -> Q Exp -> Q [(Name, Dec)] -> Q Exp
qmix qF qX qFunctions = do
              f   <- qf
              x   <- qx
              functions <- qFunctions
              return (mix f x functions)
```

5.4.2 The Function `mix`

`mix` initiates the partial evaluation process.

```
mix :: Exp -> Exp -> [(Name, Dec)] -> Exp
```

The first argument is the function to specialise. It is only specialised, if it is a construct directly representing a function: a lambda function or a call to a variable refering to a function declaration. The second ist the parameter value to specialise it to. `mix` knows that the value belongs to the first parameter of the function, all other parameters are considered dynamic. With this information, the partial evaluation of the body is initiated as if a call to the function was just unfolded.

For lambda functions, this is easy. Using `staticMatch`, the first element of the pattern list can directly be compared to the argument. With the obtained mapper, the body expression can be specialised as usual. The result is wrapped in a lambda expression for the remaining arguments.

```
mix (LamE (p:pats) e) x functions = case staticMatch p x of
       Just mapper ->
              let resBody = residual (pevalExp e mapper functions)
              in compress (LamE pats resBody)
       Nothing  -> error "Non␣exhaustive␣patterns␣in␣lambda"
```

For a variable, the function declaration has to be looked up in functions. `pevalDec` provides the the functionality for partial evaluation of a `Dec`. The only

[12] See Section 6.1 for further explanation.

difficulty is that it demands a list of arguments. Since the only given argument is the first, we artificially construct dynamic placeholders for the remaining arguments and call `Dec`. The resulting `Dec` needs to be wrapped in a lambda function to be returned as `Exp`.

6 Limitations of the Implementation

In this section, difficulties that arose while implementing the partial evaluator and their solutions are discussed. Design decisions are motivated and explained.

6.1 Obtaining Function Implementations

To unfold function calls, we need the definition of the function. This is why they need to be given as a parameter to the call of `mix`.

The problem

If a non-anonymous function is applied, it is called by name.

`AppE ... (AppE (VarE f) arg`$_1$`) ... arg`$_n$

where `f` is of type `Name` and refers to a function declaration of type `Dec`. To unfold such a function call, the declaration is needed. If it is provided per local definition by a `let` or `where`, it is contained in the same syntax tree as the call to the function and is therefore accessible during partial evaluation. If the declaration is defined outside the calling expression, somewhere else in the module or in an imported module, retrieving the declaration for the given function name is necessary.

Template Haskell offers a function `reify :: Name -> Q Info`. In theory, we can use this function to ask the compiler for information about a `Name`. If this `Name` belongs to a variable, the `Info` data is build by the constructor `VarI`.

```
data Info = ... | VarI Name Type (Maybe Dec) Fixity | ...
```

According to the documentation [15], the third argument (type `Maybe Dec`) contains the declaration which defines the variable. This would be exactly what we need: a function returning the declaration of a variable given its name. Unfortunately, although `reify` and all other arguments of the `Info` data work, the `Maybe Dec` always equals `Nothing`. It "has not yet been implemented because of lack of interest" [15].

The solution

To be able to acces the function declarations, the functions used in the expression that shall be partially evaluated have to be given as a list `[(Name, Dec)]`. When unfolding a function call, the declaration to the called name is looked up

in the list. Unfortunately, this means that every function of the partial evaluator has a parameter for the list of functions and passes it to all called functions, in case the partial evaluation of some subexpression includes unfolding.

Preparing the list of functions, we must be very careful with the names. The following example will show the pitfalls. Let `f` be a function called in an expression, that is to evaluate partially. `f` itself calls a function `g`. Thus, both functions have to be elements of the lookup list.

```
f x = g (x + 1)

g x = g (x - 1)
```

As mentioned in Section 3.3, the quotation brackets do not in any way interpret the construct between the brackets before converting it to abstract syntax. The quotation `[| f |]` thus yields the abstract syntax tree `VarE f` (as `Q` monadic computation), and not the declaration. Hence, abstract syntax trees need to be inserted into the list directly. The easiest way to keep implementation and abstract syntax of the declaration synchronised is to define the function as `Q Dec` to insert into the list and splice it in for use. Definition and splice must be in different modules, obeying the Template Haskell rules.

```
module module1 where
f_ast = [d| f x = g (x + 1) |]

module module2 where
$f_ast
```

Now, another problem arises: if the function `g` is not in scope when `f_ast` is defined, the call to `g` is not refering to `g`, but seen as an unbound variable. The quotation monad keeps track of the `Name`s in scope and generates a guaranteed fresh name. With the Template Haskell restriction of definition and splice not being in the same module, and Haskell prohibiting cyclic imports, this means we need three modules for defining `f` and `g`.

```
module module0 where

g_ast = [d| g x = g (x - 1) |]


module module1 where
import module0

$g_ast
f_ast = [d| f x = g (x + 1) |]


module module2 where
import module1
```

`$f_ast`

In general, a new module is needed for every nesting level of function calling.

To append information for `g` to the list, we need two things: the declaration as abstract syntax tree, which is directly given by `g_ast`, and the `Name`, by which `g` is called in f. The `Name` can easily be obtained using `'g`. But now, we stumble upon problem number three: In `module0`, where `g_ast` is defined, the resulting function `g` is not in scope. So, the recursive call in the definintion is not to `g'`, but another `Name`, let us call it `gdef`, generated by `Q`. The quotation brackets are smart enough to register this recursive call in `g`, so the `Name` in the function declaration and the recursive call are the same, but since the declaration is listed as tuple together with `g'`, we will not find the declaration of `g` looking up `gdef` while unfolding the recursive call. To solve this, every recursive call `VarE gdef` in the function body has to be renamed to `VarE g'`. This can be done by simply performing a depth-first search on the tree and renaming occurences. Be careful: for infinite trees, the renaming does not terminate.

All in all, three occurences of a function name must be equal:

- The name of recursive calls in the function declaration,
- the name as first component of the list pair,
- and the name of the function in the initial call of the function.

In Appendix B, the implementation of a module `FunctionList.Build` is given, which helps forcing these equalities when building the function list. It provides the functions `consDec` and `nilDec`. For a function declaration in abstract syntax, the name by which the function is called and an existing list of functions, `consDec` takes care of renaming and appends the declaration to the given list. `nilDec` gives an empty list of function declarations.

6.2 Binding Time Classification: Optimality vs. Computability

The problem of binding time classification has been explained in Section 2.3. The algorithm for binding time analysis can either be computable, that is, terminate in any case, or yield the optimal classification. This sections explains decisions that have been made and measures that have been taken for this algorithm to achieve a good classification while terminating in as much cases as possible.

The first important choice was not to expect nonterminating structures. For infinte abstract syntax trees, termination is not guaranteed. Consider as an example the following syntax tree.

```
let clause = Clause [VarP x ] (NormalB (VarE x)) []
in FunD infiniteTree (repeat clause)
```

which corresponds to

```
let clause = Clause [VarP x] (NormalB (VarE x)) []
in FunD infiniteTree [clause, clause, ...]
```

Termination is not garantueed for recursive calls leading to nontermination if interpreted either:

```
f x = f (x - 1)
```

The user is expected not to input such structures. Assuming the adherence of this restriction, the possibility of nontermination caused by the structure of the function to specialise can be ignored. Still, problematic parameters have to be taken care of. From here, this section will only speak of finite structures and "good" recursive calls.

Unfolding of function calls is the main candidate for infinite computation. Functions with no function calls should thus be safe, since nothing but unfolding extends the structure. To avoid nontermination, unfolding of a function is initiated only if at least one of the parameters of the call is static. This is implemented in `pevalExp`, checking the static flag of the arguments before calling `unfold`. If a recursive function is called with only dynamic parameters, in most cases the base case will never be entered, as the base case often is the one with concrete values. The call is reconstructed and returned as all in all dynamic. For some primarily non-recursive functions this may lead to less specialisation. In this case, termination is choosen before optimality.

For further measures that could be implemented in future to assure termination, see Section 8.

6.3 The force-evaluation Problem

Looking at the expression `2 + 1`, it seems easy to evaluate it to `3`. But for partial evaluation, the expression would be given as abstract syntax tree:

```
InfixE  (Just (LitE (IntegerL 2)))
       (VarE GHC.Num.+)
       (Just (LitE (IntegerL 1)))
```

In this representation, evaluation is not as easy. In fact, the Template Haskell syntax does not provide a possibility to evaluate such expressions. What is missing in this case is type information: Suppose a data type `Numbers`, which is an instance of the type class `Num`.

```
data Numbers = Number Integer

instance Num Numbers where
       (Number a) * (Number b) = Number (a * b)
       (Number a) - (Number b) = Number (a - b)
       (Number a) + (Number b) = Number 0
```

While multiplication and subtraction are defined as usual for numbers, the sum of two `Numbers` is always `Number 0`. The abstract syntax tree of `(Number 2) + (Number 1)` is given by

```
InfixE  (Just (AppE (ConE Number) (LitE (IntegerL 2))))
       (VarE GHC.Num.+)
       (Just (AppE (ConE Number) (LitE (IntegerL 1)))).
```

Comparing it to the abstract syntax of `Integer` addition `2 + 1`, given above, we notice that addition, `+`, is encoded to the same abstract syntax: `VarE GHC.Num.+`. The function `+` has several implementations. To reduce the syntax tree to its evaluated equivalent, we need to know which one is referred to. Since this information is not contained in the abstract syntax, evaluating the expression is much more difficult than it seems.

In most applications of Template Haskell, evaluation is not needed. The manipulation of code is done during compile time, and the produced code is compiled. While compiling, the information is available, and the splices are evaluated. In this case, manipulated syntax trees affect the further manipulation in some situations, so evaluation would be benefial.

Reducing abstract syntax is needed in this implementation of partial evaluation for two major purposes:

- The residual subexpressions can be used for binding time analysis.
- Static parameter values, that could be reduced, should still match patterns representing the reduced form.

To understand the first purpose, consider a conditional expression

```
e = if c then a else b.
```

Let `c` and `a` be static and `b` dynamic. Knowing only that `c` is static, we have to classify `e` as dynamic, since `c` could evaluate to `False` and `e = b`. If it were possible to evaluate `c` and it evaluates to `True`, we would know that `e = a` and therefore `e` is static. Another example is a guarded function body: to know which expressions are to consider, we need more information than if the guard is static: we want to know, if the condition evaluates to `True`. This issue can be summarised as follows: If we can not evaluate expressions, online partial evaluation can not perform better than offline partial evaluation, since we still only know the classification of subexpressions to decide if an expression is static, not the value of the static expression.

When matching pattern against static arguments as seen in Section 5.2.2, another problem arises: The pattern for `1` does not match the expression `2 - 1`, even if they are semantically equal!

```
-- Exp representing (2-1)
InfixE  (Just (LitE (IntegerL 2)))
       (VarE GHC.Num.-)
       (Just (LitE (IntegerL 1)))

-- Pat representing 1:
LitP (IntegerL 1)
```

This issue is even worse if unsolved. Every function calling itself recursively with a parameter reduced by a function call will never match its base case. This leads to nontermination. Partial evaluation of non-recursive functions will still terminate, but eventually yield an incorrect residual function, if functions are called with non-trivial arguments.

There is a possible solution: [17] provides a module with a function `compile`.

```
compile :: forall a . Typeable a => TExp a -> IO a
```

This functions claims to evaluate Template Haskell Expressions by compiling it. The result is written in a module and returned in the `IO` monad. Using the function `runIO`, defined in the Template Haskell library `Language.Haskell.TH` (see Section 3), the resulting `IO` computation could be converted to a computation in `Q`. The part of the program, where the module is generated, has to be omitted. The type class `Data.Typable` helps to solve the type problem described above.

This solution is not yet implemented in this work. In the next section, Section 7, presenting results, there are no test cases with functions that fail due to the force-evaluation Problem. Only function calls with elementary parameters or parameters that are built using constructors only, are used.

7 Results

This section demonstrates by two examples what can be achieved with this implementation of a partial evaluator. The first example is focussed on the possible optimisations achieved by specialising a function, while the second shows how partial evaluation can be used in image processing.

7.1 Optimisation illustrated by computing Fibonacci Numbers

In this section, the partial evaluator will be applied to a Fibonacci implementation to examine its ability to optimise computation. For suitable structures, significant improvements in execution time and space become apparent. In order to illustrate improvements achieved by partial evaluation as clearly as possible, a simple function with a large amount of computation would be convenient. A naive implementation of the Fibonacci numbers for natural numbers provides this property: As a consequence of its exponential complexity, calls with relatively simple arguments yield large computations.

7.1.1 Test setup

The natural numbers are defined as a recursive data type to avoid the force-evaluation-problem described in Section 6.3.

```
data Nat = Zero | Succ Nat
```

To compute Fibonacci numbers, an addition for natural numbers `nAdd` is needed. Its implementation can be found in Appendix C. Based on these definitions, the Fibonacci numbers can be defined by a recursive function `nFib`.

```
nFib :: Nat -> Nat
nFib Zero  = Zero
nFib (Succ Zero) = Succ Zero
nFib (Succ (Succ n)) = nAdd (nFib (Succ n)) (nFib n)
```

As described in Section 6.1, the functions `nAdd` and `nFib` are added to a list `functions` to obtain their declarations while partial evaluation.

7.1.2 Test case 1: Limited Optimisation for verbose Results

For the natural numbers from 10 to 22, we want to compare time and space used for the regular application of `nFib`, computed at runtime, to the runtime computation of the specialised application of `nFib`. For that purpose, for every $a \in \mathbb{N}$, $10 \leq a \geq 22$, a module with the following content and the required imports is created. `n` is the representation of a as `Nat`.

```
mixedFib = $( qmix [| nFib |] [| n |] functions )

fib = nFib n
```

The tests are executed using ghci 8.0.1 in the operating system Debian. ghci is called with the flag `-XTemplateHaskell` to make quotation brackets and splice operator available. Time and space measuring of computations is switched on by executing `:set +s`. If we compile every module and then execute `mixedFib` and `fib`, we obtain the compile time in seconds, and both execution time and used space for the regular computation, `fib` and the specialised computation, `mixedFib`.

Figures 2 and 3 show the comparison of `fib` and `mixedFib` for time and space measuring. Both for execution time and space used for the computation, `mixedFib` produces slightly better results, but the difference is not significant. However, observing the computations, it can be noticed that ghci spends most of the time with printing the result, instead of computing.

This test case can thus be concluded as follows: The verbose representation of natural numbers with the data type `Nat` has two disadvantages. Firstly, saving the compile time computed `mixedFib` takes nearly as much space as computing it at runtime. Secondly, the time to traverse the results of `mixedFib` and `fib`, for example for printing them, takes far more time than the actual computation. If the result of partial evaluation for a function carries more weight than the partial evaluation itself, the possibilities of optimisation by partial evaluation are limited.

7.1.3 Test case 2: Optimising Fibonacci Computation

To overcome this limit for Fibonacci computation, we need to save the precomputed Fibonacci number in a less verbose format. The solution is a function

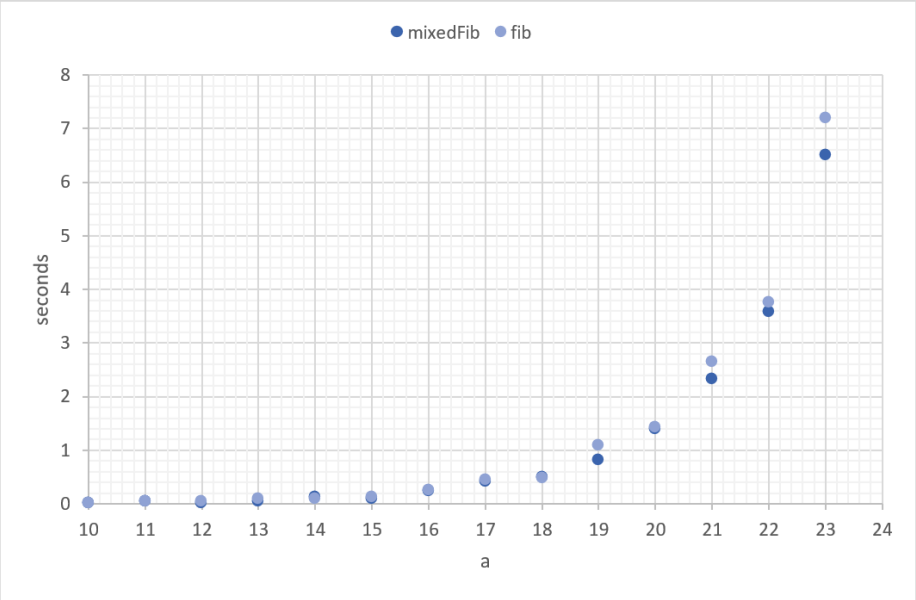

Figure 2: Execution time of specialised and unspecialised Fibonacci computation.

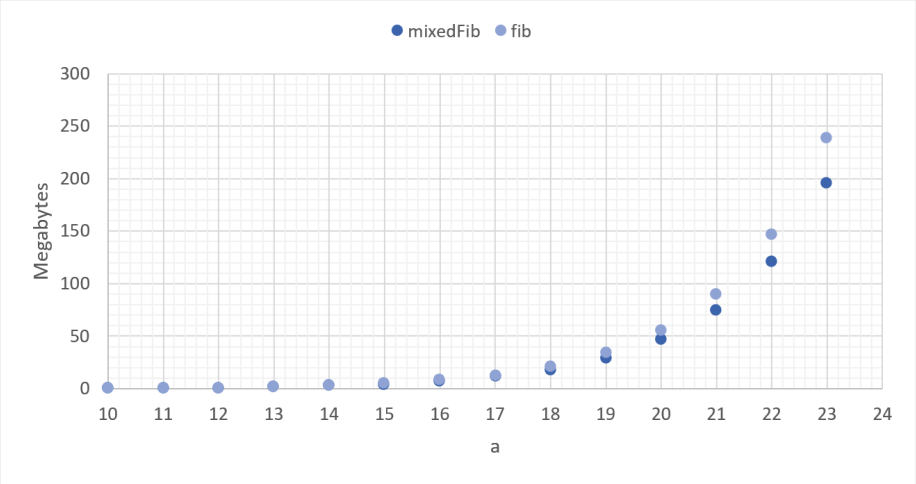

Figure 3: Space used for specialised and unspecialised Fibonacci computation.

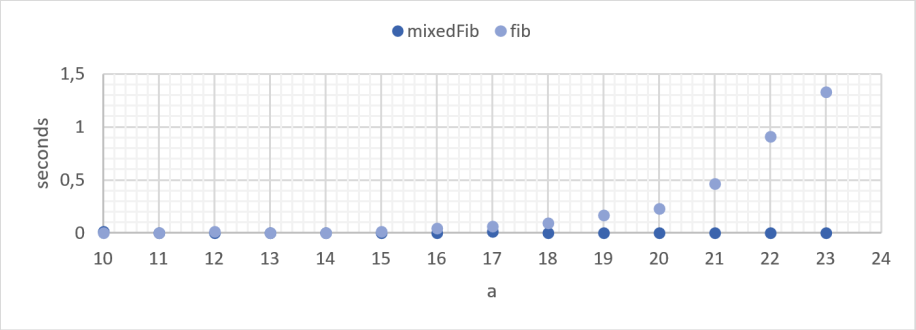

Figure 4: Execution time of Fibonacci computation saved as `Int`.

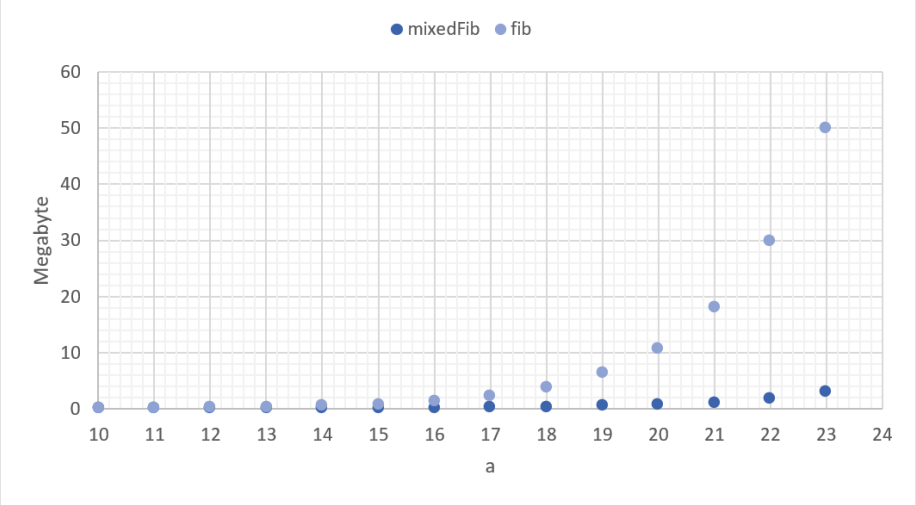

Figure 5: Space used for Fibonacci computation saved as `Int`.

`nInt`[13], which converts numbers of type `Nat` to `Integer`s. It is appended to the list `functions` for partial evaluation. The test modules are adapted as follows:

```
mixedFib = $(qpeval [| nInt |]
                    (qpeval [| nFib |] [| n |] functions)
                    functions)
fib = nInt (nFib n)
```

Execution of the tests and measuring of time and space is done exactly as in Test case 1 (see previous section: Section 7.1.2).

The results of this second attempt, visualised in Figures 4 and 5, reveal the potential of partial evaluation. While the execution time of the specialised com-

[13]Its implementation can be found in Appendix C.

putation `mixedFib` is nearly constant in our tests, the time of the unspecialised computation `fib` increases quickly. The space of the specialised computation is not a constant, since the `Integer` representation increases with the increasing input, but still represents only a fraction of the space used for the unspecialised computation. These results, especially the constant time, are impressive. Nevertheless, specialisation of a function to its only argument is more a forced interpretation at compile time, since the whole computation can be performed statically at compile time.

7.1.4 Test case 3: A Function as residual Construct

This last test case on fibonacci numbers is designed to show optimisation when specialising a function with remaining dynamic arguments. The result is a residual function, instead of a value as in the previous test cases. For the possibility of keeping one argument dynamic when specialising a function to a static argument, we need a funciton with at least two arguments.

```
nFoo :: Nat -> Nat -> Nat -> Nat
nFoo a b = nAdd (nFib a) (nFib b)
```

The function `nFoo` computes the Fibonacci number for its two arguments and returns the sum. It is added to the list `functions`. The module is adapted as follows.

```
mixedFoo = $(qpeval [| nFoo |] [| n |] function)

foo = nFoo n
```

Specialising `nFoo` to its first argument yields a function `Nat -> Nat`. `mixedFoo` thus needs to be applied to another natural number represented as `Nat`. For the tests, we compile the module for $18 \leq a \geq 31$ (`n` the `Nat` representation of a as above) and apply `mixedFoo` and `foo` to `Succ Zero`, measuring again execution time and space used for the computation. For simplicity, the result is converted to an `Integer`, which takes place at run time for both the specialised and the unspecialised call. The following is an example call to `mixedFoo` and `foo` for a module compiled with $a = 20$.

```
nInt (mixedFoo (Succ Zero))  ⇝ 10947
nInt (foo (Succ Zero))  ⇝ 10947
```

Figures 6 und 7 visualises the results. While for small a, the performance of `foo` and `mixedFoo` is comparable (due to the complexity of the residual function `mixedFoo`), the unspecialised computation `foo` needs up to four times more time and up to ten times more space than specialised `mixedFoo`.

7.2 Application: Specialising Matrix Multiplication

In this section, we specilise matrix multiplication to fixed entries in the first matrix and a second matrix of fixed size. This process has many applications:

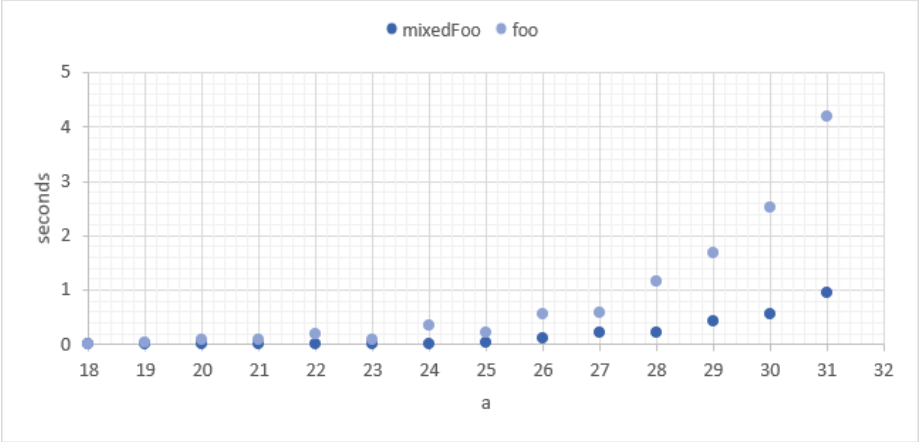

Figure 6: Execution time of computation with two arguments.

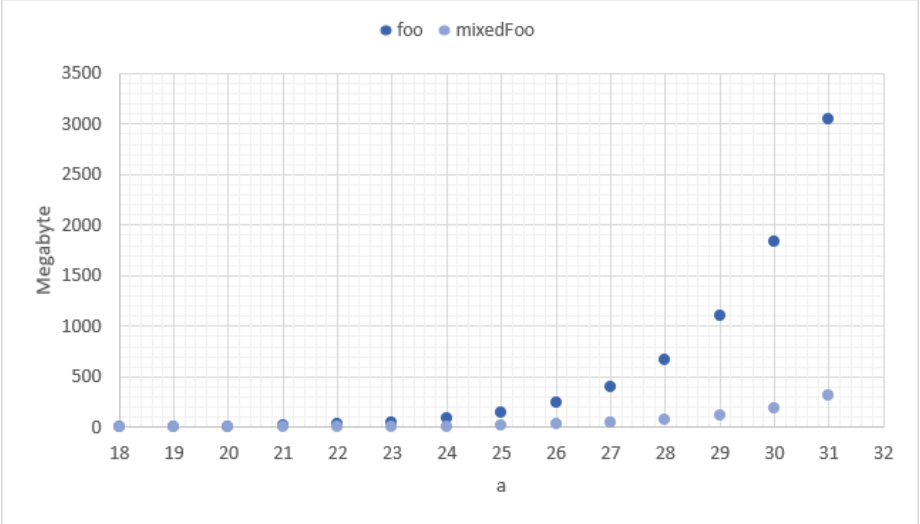

Figure 7: Space used for computation with two arguments.

Matrix multiplication is used in numerous areas of computer science, from scientific computing to image processing. One example is *Descrete Cosine Transform*. An image is split into fragments of equal size that are interpreted as matrices. For particular matrices A, A', the Discrete Cosine Transform is composed of a fragment C for every fragment B in the original image. C is computed from B, A and A' by matrix multiplication [18].

$$C := ABA'$$

Since A and A' are fixed values for all C, which are known before computing the fragments, we can use partial evaluation to specialise the multiplication to A and A'. As the result, we expect a residual function with only one parameter for B. This function can be applied to all fragments, performing the residual computations, while all shared computation is only done once, at compile time.

We will now apply partial evaluation to a matrix multiplication AB with A, $B \in \mathbb{Z}^{2\times 2}$ to a fixed value for A:

$$A = \left(\begin{smallmatrix} 1 & 1 \\ -1 & 2 \end{smallmatrix}\right)$$

The result of multiplying A by $B = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ is a matrix $C = \left(\begin{smallmatrix} a+c & b+d \\ -a+2c & -b+2d \end{smallmatrix}\right)$. We expect the residual function obtained by partial evaluation to assign arbitrary values a, b, c, $d \in \mathbb{Z}$ to the matrix C.

7.2.1 Mathematical Definition of Matrix Multiplication

For matrices $A \in K^{m\times n}$ and $B \in K^{n\times p}$ (where K is a ring),

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}, B = \begin{pmatrix} b_{11} & \dots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{np} \end{pmatrix},$$

the product $C = AB$ is defined by $C^{m\times p}$,

$$C = \begin{pmatrix} c_{11} & \dots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \dots & c_{mp} \end{pmatrix},$$

with $c_{ij} = \sum_{k=1}^{n} 1_{ik} b_{kj} \quad \forall (i,j) \in \{1,..m\} \times \{1,..p\}$.

7.2.2 Implementation of matrix multiplication

In Haskell, the multiplication of two matrices $A \in \mathbb{Z}^{m\times n}$ and $B \in \mathbb{Z}^{n\times p}$ can implemeneted as follows.

```
mm :: [[Int]] -> [[Int]] -> [[Int]]
mm _ [] = []
mm rowVecs (c:columns) = mmColumn rowVecs c : (mm rowVecs columns)
```

```
mmColumn :: [[Int]] -> [Int] -> [Int]
mmColumn [] column = []
mmColumn (r:rows) column
       = dotProduct r column : (mmColumn rows column)

dotProduct :: [Int] -> [Int] -> Int
dotProduct [] [] = 0
dotProduct (x:xs) (y:ys) = r*c + (dotProduct row column)
dotProduct _ _ = error $
    "Number␣of␣entries␣must␣be␣equal␣to␣calculate␣the␣dot␣product!"
```

A and B are represented by lists of lists. To keep the example implementation simple, we assume that A is represented as a list of row vectors, while B and the result are represented as a list of column vectors (the representations as a list of row vectors and as a list of column vectors an easily be transformed into each other with the predefined function `Data.list.transppos`). All three functions are defined recursively. Note: the recursion in `mm` is controlled by A, the recursion in `mmColumn` is controlled by B, and the recursion in `dotProduct` is controlled by both arguments.

An example call yields correct result according to the mathematical definition in Section 7.2.1.

```
mm [[1,1], [-1,2]] [[1,3], [2,4]] ⇝ [[4,5], [6,6]]
```

The functions `mm`, `mmColumn` and `dotProduct` are prepared for partial evaluation in a list named `functions` (see Section 6.1 for further explanation of the list).

7.2.3 Test

The function `mm` is now specialised to A as its first argument.

The implementation works for matrices of arbitrary sizes, i.e. the values for m, n and p are not fixed by `mm`. If we specialise to the first argument, m and n are given by the size of the parameter value A, but p is not. The number of colums of the dynamic argument B thus can not be determined during partial evaluation. As a consequence, the recursion in `mm` and `dotProduct` can not be completely determined either. The recursive calls can not be specialised to a single function body, since two clauses match the static parameter. Instead, a residual declaration containing two clauses is returned and the result of partial evaluation is not a matrix to plug in values, but a more complex function declaration.

However, p *is* fixed in our example, since B is known to be a matrix in $\mathbb{Z}^{2\times 2}$, just as it it is known in Discrete Cosine Transform, since the image is split in fragments of equal size. To use this extra information, we trick the partial evaluator: We specialise `mm` not only to the static matrix A, but to a matrix containing variables for the entries of B, too. The result is the matrix C described previously, as desired.

To input both arguments at once, we use the uncurried version of `mm`.

```
mm :: ([[Int]], [[Int]]) -> [[Int]]
```

This is necessary to control unfolding of the recursive calls in `mm`, which depends on the second argument. Besides, specialising a function to all of its arguments at once is more efficient than computing a residual function for each argument and then specialising it to the subsequent.

Partial evaluation of `mm` to values for A and variables for B yields a plain matrix instead of a function with an argument for B, since the only parameter is considered static. The residual matrix contains the given variables for the entries of B. To use it for different matrices B, we wrap the residual matrix in a function with a pattern matching the entries of B and assigning them to the variables in the residual matrix.

In order to generate capturable variables for the values in B, the abstract syntax is created directly instead of quoting the concrete syntax. The naming can be supervised by using the function `mkName :: String -> Name`. It generates a name that is capturable by the given `String`. A variable is build by applying the constructor `VarE`.

```
a = VarE (mkName "a")
b = VarE (mkName "b")
c = VarE (mkName "c")
d = VarE (mkName "d")
```

The concrete syntax variables corresponding to this abstract syntax can be captured by the concrete syntax patterns `a`, `b`, `c` and `d`. `qPair` constructs a matrix B containing the capturable variables, quotes the value for A and combines both to the input pair for partial evaluation.

```
qPair :: Q Exp
qPair = do
       let varMatrixB = ListE [ListE [a,c], ListE [b,d]]
       valMatrixA <- [| [[1,1], [-1,2]] |]
       return (TupE [valMatrixA, varMatrixB]
```

With all this preparations done, we can now initiate partial evaluation, splice the result and wrap it in a function `mixed`.

```
mixed a b c d = $( qmix [| mm |] qPair functions )
```

The ghc flag `-ddump-simpl` prints the content of a loaded module after simplifications have been applied by the compiler. It can thus be used to verify that the splice produces the expected result. If it suffices to check the splices before they are simplified by the compiler, the flag `-ddump-splices` can be used. It prints the unsimplified splices in a readable, less verbose format than `-ddump-simpl`. When compiling the test module, `-ddump-splices` prints the following result for `mixed` (module names are omitted for readability):

```
((((1 * a) + ((1 * c) + 0))
```

```
       : (((negate 1) * a) + ((2 * c) + 0)) : [])
:
(((1 * b) + ((1 * d) + 0))
       : (((negate 1) * b) + ((2 * d) + 0))) : []
:
[])
```

This output represents the expected matrix $C = \left(\begin{smallmatrix} a+c & b+d \\ -a+2c & -b+2d \end{smallmatrix}\right)$. The remaining computation is a combination of basic arithmetic and the list constructors `:` and `[]`. The recursion has been unrolled successfully.

At runtime, `mixed` can be used to multiply an arbitrary matrix $B = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ by $A = \left(\begin{smallmatrix} 1 & 1 \\ -1 & 2 \end{smallmatrix}\right)$.

```
mixed 1 2 3 4 ⇝ [[4,5], [6,6]]
```

```
mixed 1 2 3 4 == mm [[1,1], [-1,2]] [[1,3], [2, 4]] ⇝ True
```

Partial evaluation has thus succesfully been applied to matrix multiplication. The function `mm` has been specialised to the matrix `[[1,1], [-1,2]]`. The computation left for the residual function `mixed` is minimal: the variables in the residual function are replaced, the entries are calculated by using elementary arithmetic, and the list is constructed. `mixed` can be used for arbitrary matrices $B \in K^{n \times p}$. The same process of specialisation can be achieved for any other matrices A by replacing the values in the definition of `valMatrixA` in `qPair`.

8 Next Steps in Implementation

This section lists features that can be implemented in an advanced version of the partial evaluator.

Integrate module `Compile.hs` to solve the force-evaluation problem. Section 6.3 discusses the problem of reducing abstract syntax by evaluating the represented expression. A possible solution is provided by the module `Language.Haskell.TH.Compile.hs`, which performs the reduction by compiling expressions.

Before integrating it, the module has to be adapted. The function `compile` accepts an `Exp` and returns a module with the compiled code. While the code producing the module can easily be removed to return a plain expression, the basic problem remains: The function's return value can not be predicted before calling the function, since the abstract syntax data type `Exp` encodes all types of expression.

```
[| True :: Bool |] :: Q Exp
[| 4 :: Integer|] :: Q Exp
```

The force-evaluation problem prevents this implementation from solving three important tasks:

- Select matching guards in a guarded body
- Decide for conditionals wether the `then` case or the `else` case will be applied
- Match non-trivial static arguments to function patterns

In conditions and guards, the expression to compile has to be a `Bool`, so the problem can be solved by making `compile` return a `Bool`. For pattern matching, the static argument would have to be evaluated before matching a pattern. `Pat` and `Exp` can not be compared directly. As a consequence, a fixed return type can not be determined. Hence, the modules semantics have to be altered to perform the pattern matching while compilation, and return the result in a fixed format.

Implement missing constructors. Although Template Haskell provides abstract syntax for every construct in Haskell, not all of them can be specialised yet. The function `pevalExp` is not implemented for all constructors of `Exp`. The constructors represent kinds of nodes in the abstract syntax tree, e.g., the constructor `TupE` provides a representation for tuples. As seen in Section 5.3, `pevalExp` is implemented for `TupE`, hence, tuples can be evaluated, as long as the components are expressions that can also be evaluated. Another example of a constructor for `Exp` is `CaseE`, providing abstract syntax trees for `case of` statements. It is one of the constructors that `pevalExp` is not yet implemented for, so `case of` statements can not be specialised yet. This does not limit the capabilites of the partial evaluator, since unimplemented constructs can be expressed using implemented constructs. For example, a `case of` statement is equivalent to a call to a function with corresponding patterns.

A full list of unimplemented constructors can be found in Appendix D.

Currently, these cases of `pevalExp` are implemented as `undefined`. This is usefull for implementation and testing, since `undefined` throws an exception with an appropriate error message, if evaluated. For the actual use of the partial evaluator, it might be reasonable to return the unaltered input expression as dynamic instead. Like this, specialisation can be performed for all possible constructs without being prevented by the exception.

Save unfolded calls with static parameters. Concrete values for the dynamic parameters do not influence the specialisation of a function. The residual body depends only on the classification of arguments in static and dynamic and on the values given for the static parameters. Every call of a function with the same static arguments is specialised to the same residual function body. Hence, if unfolding an application of `f` to static parameters `s` and dynamic parameters $\texttt{d}_1$ leads to another function application of `f` to the same static parameters `s` and potentially different dynamic parameters $\texttt{d}_2$, unfolding those calls again and again results in nontermination. To avoid this, a list of unfolded calls with a list of the static parameters can be passed recursively to the children nodes when specialising a syntax tree.When unfolding an application of `f` to `s` and `d`,

earlier unfolds of `f` can be looked up in the list. The static parameters `s` can be compared syntactically to the static parameters of eventual earlier calls. If an equivalent call is found, unfolding is not performed. As an example, consider a call `f 4 y` to the function

```
f x y = if y == 0
          then x
       else
          f x (y - 1)
```

The first argument `x` is static, the second argument `y` is dynamic. The call is unfolded to the expression

```
if y == 0
   then 4
else
   f 4 (y - 1)
```

now, further specialisation of the expression is initiated by calling `pevalExp`. As additional information, we pass to `pevalExp` a pair `(f, [(True, 4), (False, y)]` to imply, that `f` already has been unfolded with the static value `4` as first argument `x` and a dynamic second argument. If `pevalExp` reaches the expression `f 4 (y - 1)`, it consideres to unfold the call, which would obviously lead to nontermination if unfolded again and again, since the consecutive recursive calls do not yield new information. And here, the information about previous unfolds takes effect: The static argument `4` is compared to the static argument of the previous unfold. Since they match, the function should not be unfolded again.

Add function `dynamicMatch` to handle partially static structures while unfolding. As described in Section 5.2.2, patterns corresponding to dynamic parameters are used to rename the whole parameter value without further checks. Using this rather simple technique, a call to the function `fst (a,b) = a` with a tuple `(4,m)` (where `4` is static and `m` is dynamic) would be specialised to

```
let (a,b) = (4,m) in a
```

and annotated as dynamic. Looking at this example, it is obvious that the expression could be static, since the body of the function `fst` accesses the static component of the pair only. The desired specialisation yields `4`.

To achieve this, we need a function `dynamicMatch`. Following the structure of the function `staticMatch`, it computes a mapping of variable names to corresponding static expressions, in the above example `[(a,4)]`. For the dynamic parts of partially static structures, above, this is `m`, it computes the renaming declaration as before. Plugging in `dynamicMatch` in the pattern matching functionality (see 5.2.2), this mapper can be combined with the mapper returned by `staticMatch` and used to perform partial evaluation on the body expression. With the mapping containing all static parts, `a` can be replaced by `4` in the above example. For the obtained static body, the renaming of dynamic arguments is skipped as unnecessary, yielding `4` as specialised expression.

Add support for functions without explicit patterns. The current implementation can not handle functions defined without giving an explicit pattern for its parameters. For example, the function

```
f = map (+2)
```

can not be specialised, since the pattern, that is checked for unfolding, is empty. To specialise the function `f`, a pattern has to be added..

```
f xs = map (+2) xs
```

To support specialisation of the former kind of function definition, the unfolding functionality (see Section 5.2) has to be adapted.

Check declarations and arguments by signature. The current implementation just assumes that it is given valid inputs. If that assumption does not hold, partial evaluation fails, often with non-descriptive error messages. Using the information of available signatures, two possibilities of invalid arguments could be detected:

- The patterns of different clauses of a function declaration could be compared to the signature. This would prevent the user from specialising functions like

  ```
  f 1 = 0
  f "k" = 1
  ```

- The arguments of function application could be typechecked against the pattern. For example, `f "eins"` would not be possible for a function `f :: (String, Integer)`.

Implement interface for specialisation to more than one static argument. The code for partial evaluation is already able to specialise to several static arguments, since unfolding function calls is possible for a list of static arguments. What remains to do is to adapt the user interface `qmix` and the function `mix`. `mix` invokes partial evaluation by calling the unfolding functionality.

If a function is to be specialised to several of its parameters, the specialisation process is more efficient if specialising to all arguments at once, since no intermediate partial functions have to computed and returned. The overhead of the computations to initiate partial evaluation in the function `mix` is reduced.

9 Conclusion

This work provides an overview of partial evaluation as a specialisation technique. The theoretical foundation, given by Kleene's s-m-n theorem, has been explained. Basic procedures, like the classification of parameters and expressions

into static and dynamic (called binding time analysis), and online and offline partial evaluation, have been discussed. The theoretical possibility of compilation and compiler generation by selfapplication of partial evaluation combined with an interpreter has been introduced in terms of the Futamura projections.

An important requirement for an implementation of a partial evaluator for a certain language is the possibility of metaprogramming, i.e. the possibility to manipulate and generate programs in this language as program data. The language extension Template Haskell allows type safe compile time metaprogramming for the functional language Haskell. This work has provided an overview of the functionality of Template Haskell and introduced the language components that are used in the implementation of a partial evaluator.

The implementation of a partial evaluator specialising Haskell functions was succesful. The prototype uses Template Haskell to provide a function `qmix` as interface for the user. `qmix` can be applied to a function and a value for its first parameter to obtain a residual version of the original function, specialised to the given value. Partial evaluation includes binding time analysis, reduction of function bodies to simple expressions and unfolding of function calls by replacing a call by the body of the function, if possible.

The optimisation achievable by partial evaluation strongly depends on input function and argument. For some inputs, partial evaluation does not yield a more efficient residual function. This work presented two applications in which partial evaluation is of great benefit.

`qmix` has succesfully been applied to Fibonacci computations. Especially in matters of execution time, the results reveal great benefits of partial evaluation: the runtime complexity of Fibonacci computation has been reduced to a constant by executing the whole computation at compile time. In a function with remaining dynamic expressions (a function including computations that cannot be precomputed at compile time) the speedup is still significant. In the conducted tests, the specialised constructs perform in average in 20% of the time used by the corresponding unspecialised computations.

The application of the prototype to matrix multiplication illustrates a real world application of partial evaluation. In order to compress an image it is split into fragments of fixed size, which are multiplied by a matrix with fixed values. Partial evaluation is able to specialise matrix multiplication to the matrix with fixed values and precompute the shared computation of all multiplications, resulting in a parameterised matrix. For the single fragments, the remaining computation is minimal: the entries of a single fragment of the image replace the variables in the residual matrix and basic arithmetic is used to calculate the values.

Despite this encouraging results, a lot of work remains to be tone before the implemented partial evaluator can be used to specialise real world problems. Termination of the partial evaluation process is not yet assured. As a simple way of forcing termination, computation could be cut by not performing further unfolds when reaching a certain limit. However, under the assumption of reasonable input (no functions with infinite representations or functions representing infinite computation, no infinite static values), better approaches to support

termination are imagineable. The aim is, not to prevent partial evaluation from exploring the function in sufficient detail for efficient residual functions.

The current implementation does not yet support specialisation of every construct possible in Haskell. To provide this, some further cases have to be implemented. Besides, a lot of small improvements could be realised to yield more concrete error messages, better usability and simpler residual constructs.

In conclusion, partial evaluation is a powerful technique. In this work, a promising prototype of a partial evaluator for Haskell has been presented. After further developement, the partial evaluator could be used to specialise abstract programs to solve problems more efficiently.

Appendices

A Helper Functions

```
unionMaybe :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
unionMaybe f (Just u) (Just v) = Just (f u v)

valDec :: Pat -> Exp -> Dec
valDec pat exp = ValD pat (NormalB exp) []

tupConcat :: ([a], [b]) -> ([a], [b]) -> ([a], [b])
tupConcat (a1, b1) (a2, b2) = (a1 ++ a2, b1 ++ b2)

allStatic :: [(Bool, a)] -> Bool
allStatic = (foldr1 (&&)) . (map fst)

residuals :: [(Bool, a)] -> [a]
residuals = map snd

compress :: Exp -> Exp
compress (LetE [] x) = x
compress (LamE [] x) = x
...
compress other = other
```

B Module `FunctionList.Build`

```
module FunctionList.Build where

import Language.Haskell.TH

consDec :: Q [Dec] -> Name -> Q [(Name, Dec)] -> Q [(Name, Dec)]
consDec qf name qlist = do
    list <- qlist
    [signgature, function] <- qf
    let renamedFunction = renameDec function name
        pair = (name, renamedFunction)
        newList = pair:list
    return newList

nilDec :: Q [(Name, Dec)]
nilDec = return []

renameDec :: Dec -> Name -> Dec
```

```
renameDec (FunD oldN clauses) newN
       = FunD newN (map (renameInClause oldN newN) clauses)

renameInDec :: Name -> Name -> Dec -> Dec
renameInDec oldN newN (FunD name clauses)
       = FunD name (map (renameInClause oldN newN) clauses)

renameInClause :: Name -> Name -> Clause -> Clause
renameInClause oldN newN (Clause ps body ds)
       = Clause ps (renameInBody oldN newN body) ds

renameInBody :: Name -> Name -> Body -> Body
renameInBody oldN newN (NormalB exp)
       = NormalB (renameInExp oldN newN exp)

renameInExp :: Name -> Name -> Exp -> Exp
renameInExp oldN newN (VarE name)
    | name == oldN = VarE newN
    | otherwise = VarE name
renameInExp oldN newN (ConE name) = ConE name
renameInExp oldN newN (LitE lit) = LitE lit
renameInExp oldN newN (AppE e1 e2)
    = AppE (renameInExp oldN newN e1) (renameInExp oldN newN e2)
renameInExp oldN newN (InfixE mE1 op mE2) =
       let newME1 = renameInMaybeExp oldN newN mE1
            newME2 = renameInMaybeExp oldN newN mE2
            newOp = renameInExp oldN newN op
       in InfixE newME1 newOp newME2
renameInExp oldN newN (UInfixE e1 op e2) =
       let newE1 = renameInExp oldN newN e1
           newE2 = renameInExp oldN newN e2
           newOp = renameInExp oldN newN op
       in UInfixE newE1 newOp newE2
renameInExp oldN newN (ParensE e) =
       ParensE $ renameInExp oldN newN e
renameInExp oldN newN (LamE ps e) =
       LamE ps $ renameInExp oldN newN e
renameInExp oldN newN (LamCaseE ms) = undefined
renameInExp oldN newN (TupE es) =
       TupE $ map (renameInExp oldN newN) es
renameInExp oldN newN (UnboxedTupE es) = undefined
renameInExp oldN newN (CondE i t e) =
       let newI = renameInExp oldN newN i
           newT = renameInExp oldN newN t
           newE = renameInExp oldN newN e
       in CondE newI newT newE
```

```
renameInExp oldN newN (MultiIfE _) = undefined
renameInExp oldN newN (LetE decs e) =
       LetE (map (renameInDec oldN newN) decs) (renameInExp oldN newN e)
renameInExp oldN newN (ListE es) =
       ListE (map (renameInExp oldN newN) es)


renameInMaybeExp :: Name -> Name -> Maybe Exp -> Maybe Exp
renameInMaybeExp oldN newN (Just e) =
       Just (renameInExp oldN newN e)
renameInMaybeExp _ _ Nothing = Nothing
```

C Data Type `Nat`

```
nAdd :: Nat -> Nat -> Nat
nAdd Zero y = y
nAdd (Succ x) y = Succ (nAdd x y)

nInt :: Nat -> Integer
nInt Zero = 0
nInt (Succ x) = 1 + (nInt x)
```

D Missing Constructors

The function `pevalExp` is undefined for some constructors c of the data type `Exp`:

```
pevalExp c mapper functions = undefined
```

The following constructors are concerned.

- `InfixE (Maybe Exp) Exp (Maybe Exp)`
- `UInfixE Exp Exp Exp`
- `LamCaseE [Match]`
- `UnboxedTupE [Exp]`
- `MultiIfE [(Guard, Exp)]`
- `LetE [Dec] Exp`
- `CasE Exp [Match]`
- `DoE [Stmt]`
- `CompE [Stmt]`
- `ArithSeqE Range`

- `RecConE Name [FieldExp]`
- `RecUpdE Exp [FieldExp]`
- `StaticE Exp`

The function `pevalGuard` is unimplemented for the constructor `PatG [Stmt]` of type `Guard`.

References

[1] S. Dal-Zilio, "A self-applicable partial evaluator for a subset of Haskell." August 1993.

[2] D. Coutts, "Partial Evaluation for Domain-Specific Embedded Languages in a Higher Order Typed Language." October 2004.

[3] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[4] R. C. Boolos, George S. Jeffrey, *Computability and Logic*. New York, NY, USA: Cambridge University Press, 1987.

[5] S. C. Kleene, *Introduction to Metamathematics*. Wolters-Noordhoff, 1952.

[6] J. Hatcliff, "An Introduction to Online and Offline Partial Evaluation Using a Simple Flowchart Language," in *Partial Evaluation* (J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, eds.), (Berlin, Heidelberg), pp. 20–82, Springer Berlin Heidelberg, 1999.

[7] T. Æ. Mogensen, "Partial evaluation: Concepts and applications," in *Partial Evaluation* (J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, eds.), (Berlin, Heidelberg), pp. 1–19, Springer Berlin Heidelberg, 1999.

[8] M. Sperber, "Self-applicable online partial evaluation," in *Partial Evaluation* (O. Danvy, R. Glück, and P. Thiemann, eds.), (Berlin, Heidelberg), pp. 465–480, Springer Berlin Heidelberg, 1996.

[9] L. A. Lombardi and B. Raphael, "LISP as the language for an incremental computer," in *The Programming Language Lisp: Its Operation and Applications*, pp. 204–219, MIT Press, 1964.

[10] Y. Futamura, "Partial evaluation of computation process–an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, vol. 12, pp. 381–391, Dec 1999. Updated version: Originally published in "Systems.Computers.Controls" in 1971.

[11] J. Jørgensen, "Generating a compiler for a lazy language by partial evaluation," in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, (New York, NY, USA), pp. 258–268, ACM, 1992.

[12] N. D. Jones, P. Sestoft, and H. Søndergaard, "An experiment in partial evaluation: The generation of a compiler generator," in *Proc. Of the First International Conference on Rewriting Techniques and Applications*, (Berlin, Heidelberg), pp. 124–140, Springer-Verlag, 1985.

[13] T. Sheard and S. P. Jones, "Template Meta-programming for Haskell," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[14] "HaskellWiki on Template Haskell." `https://wiki.haskell.org/Template_Haskell`. Accessed: 2019-01-01.

[15] "Package Language.Haskell.TH Haddock Documentation." `https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html`. Accessed: 2019-01-01.

[16] "Glasgow Haskell Compiler User's Guide: Template Haskell." `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#template-haskell`. Accessed: 2019-01-01.

[17] "Language.Haskell.TH.Compile.hs." `https://github.com/joom/th-new/blob/master/Language/Haskell/TH/Compile.hs`. Accessed: 2019-01-01.

[18] "Discrete Cosine Transform." `https://en.wikipedia.org(wiki/Descrete_cosine_transform`. Accessed: 2019-01-06.

Ich versichere hiermit, dass ich die Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 7. Januar 2019

Klara Schlüter