CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Lazy to Strict Language Compiler

Master's thesis in Computer Science and Engineering

PHILIP THAM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

A Lazy to Strict Language Compiler

PHILIP THAM

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

A Lazy to Strict Language Compiler
PHILIP THAM

© PHILIP THAM, 2017.

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Carlo A. Furia, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Department of Computer Science and Engineering
Gothenburg, Sweden 2017

A Lazy to Strict Language Compiler
PHILIP THAM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The evaluation strategies of programming languages can be broadly categorised as strict or lazy. A common approach to strict evaluation is to implement a call-by-value semantics that always evaluates expressions when they are bound to variables, while lazy evaluation is often implemented as call-by-need semantics that evaluates expressions when they are needed for some computation. Lazy semantics makes use of a data structure called thunk that contains an expression, whose evaluation has become suspended, together with its environment. This thesis presents (1) a Haskell definition of the existing semantics of CakeML, a strict programming language, (2) a Haskell definition of a lazy semantics for the pure part of CakeML, and (3) a Haskell implementation of a compiler that compiles lazy CakeML to strict CakeML as defined in (1) and (2). The compiler makes use of stateful features in strict CakeML to optimise evaluation so that each thunk is evaluated at most once, simulating a call-by-need semantics.

Acknowledgements

I would like to thank Magnus Myreen for being my supervisor, giving me invaluable advice and support. I would also like to thank Carlo Furia for being my examiner and providing very helpful feedback on this report; Anton Ekblad for his help with general questions about Haskell and compilers. Finally, I would like to thank my fellow computer scientists in our lunch room Monaden for the great time that we spent together these past few years.

Philip Tham, Gothenburg, June 2017

Contents

1

Introduction

Lazy evaluation is a style of programming where expressions that are bound to variables are not evaluated until their results are needed in another part of the program [1]. This is often put in contrast with another evaluation style called strict evaluation, which always evaluates expressions immediately as they are to be bound to variables.

The main difference between lazy and strict programming languages is the steps of evaluation that their respective semantics take. With strict semantics, expressions are evaluated when they are created, while with lazy semantics, expressions are evaluated when they are needed [2]. This means that in lazy semantics you can give a function an argument whose evaluation does not terminate, and as long as that argument is never used, the function itself can still terminate normally.

It has been argued that lazy programming (and functional programming in general) brings improvement to software development in the form of modularity [3] [4]. Laziness allows the creation of certain types of programs that cannot be executed by using strict semantics. For example, the *Fibonacci numbers* is an infinite sequence of numbers where each number is equal to the sum of the two previous numbers. Defining the list of all Fibonacci numbers as an expression with strict semantics would cause the execution to never terminate, as it is required for the expression to be fully evaluated when assigning it to a variable. In a lazy context, however, the expression that defines the list of all Fibonacci numbers would simply not be evaluted until it is actually needed, meaning that simply assigning it to a variable would not cause the program to not terminate.

While lazy evaluation can be seen as beneficial, there are cases where it is not optimal. For example, programs where there are no superfluous expressions would not have much use of lazy evaluation. When it is known that all expressions bound to variables will be used, their evaluation is inevitable. The underlying reason to why it would be worse to apply lazy evaluation, rather than strict evaluation, is how lazy evaluation delays the evaluation of expressions. The act of delaying evaluation creates an overhead in both memory and processing time. In cases where there are no superfluous expressions in the

code, using lazy semantics can cause slower evaluation time when compared to strict semantics. Despite this potential drawback, laziness can be a desirable feature in many use cases.

1.1 An example of lazy vs strict evaluation

In order to paint a better picture of the benefits of using lazy evaluation, this section will illustrate how lazy and strict evaluation differ by using an example.

A useful feature in lazy programming languages is infinite data structures, such as infinite lists. Using these infinite data structures will not cause non-termination, as long as the program is not trying to access all of its elements. In Haskell, a lazy functional language, there exists a function `take n list` that takes the first `n` elements from `list` and returns a new list with these elements. It is possible to write `take 5 [1..]` and get the result `[1, 2, 3, 4, 5]`, even though `[1..]` is an expression of an infinite list. This is because only the used part of the list is generated. Below is an example of how `take` could be implemented. The function takes an integer (for the number of elements to take) and a list as arguments. This assumes that there is a datatype definition for `List` with two constructors:

- `Cons (a, List as)` for when there is an element `a` and the rest of the list `as`
- `Nil` for the empty list

```
take n list =
  if n == 0 then
    Nil
  else
    case list of
      Cons (a, as) -> Cons (a, (take n-1 as))
      Nil          -> Nil
```

The function can be written like this (with some syntax differences) for any language, but the semantics changes the strategy of evaluation. For example, a strict language will evaluate all of `list` when pattern matching, while a lazy language is likely to evaluate only part of the list as much as needed.

The evaluation strategy of strict semantics is restrictive e.g. in the way that infinite data structures cannot be used. The example above will not terminate with strict semantics, as the evaluation of the infinite list will end up in an infinite loop. This thesis will explore the connection between lazy and strict semantics and how this connection can be used to allow lazy evaluation in a strict language.

2

Goals and limitations

This chapter will describe the goals and limitations of this thesis. The project will first be described in terms of tasks that it was divided into. This is followed by the limitations that were imposed on the project in order for it to be finished within the given time frame.

2.1 Project description

The goal of this project is to create a compiler that takes code from a lazy programming language and compiles it to a target language that has strict evaluation. The resulting code should have the same user-observable semantics as the given lazy code. The purpose of creating such a compiler is to explore the formal connection between lazy and strict evaluation. The compiler will be written in the programming language Haskell. The source language for the compiler will be a lazy definition of the semantics of CakeML, a strict functional language. The target language to which the compiler will output will be CakeML, with its original strict semantics. This means that the semantics to express laziness will be defined as part of this project and used as a base to implement the compiler.

Internally, the compiler will take a CakeML expression with lazy semantics and translate it to one or more strict CakeML expressions in order to get a semantically equivalent evaluation. When it comes to handling laziness, lazy expressions will be translated to explicit thunk values [5] in CakeML. Thunks are expressions that have yet to be evaluated. Thunk values are not primitive in CakeML. The compiler will create code that implements the thunk values in CakeML. The compiler will target the stateful features of CakeML to avoid repeated evaluation of the same thunk values.

The operational semantics of lazy CakeML will be written as a part of the project. As the semantics of CakeML is specified in higher-order logic (HOL) [6], the operational semantics for CakeML will be translated from HOL to Haskell functions. The compiler

will then connect the lazy and strict semantics.

Tests will be written to show that the compiler generates semantically equivalent code. This project will employ manual unit testing to test expressions and steps of evaluation.

In summary, the project consists of three major parts that were developed in parallel:

- Writing lazy semantics for the programming language CakeML (Chapter 5)
- Creating a compiler that translates the lazy semantics to the original strict semantics of CakeML (Chapter 6)
- Testing the compiler by writing operational semantics in the form of functions and running unit tests. (Described in various chapters)

The artifacts are available in `https://github.com/th4m/masterthesis`, as well as in the appendices of this thesis. Appendix A contains the Haskell definition of CakeML's abstract syntax. Appendix B contains various datatypes used to define the semantics. Appendix C contains the code for the compiler, including helper functions.

2.2 Limitations

Due to the time constraints of this thesis, certain limitations will be enforced to keep the workload within a reasonable size. These limitations are:

Only expressions. As expressions make up of the majority of the language, they will be the topmost construct of the input language's grammar that will be implemented. Expressions allow operations and values to be represented, which is important for a programming language. This limitation means that declarations and modules will not be a part of this project.

No parser. No actual parser that takes written CakeML code was created for this thesis. Instead, an internal representation of CakeML expressions was created in Haskell and used to create the semantics and compiler. A parser that takes CakeML code in string format and converts it to the internal representation can be created in the future. It should not prove to be a challenging task to link the parser together with the compiler.

Specific version of CakeML. At the time of writing, the programming language CakeML is still evolving, meaning that the semantics of the language is prone to change at any time. In order to keep a steady workflow, any updates to the semantics after the start of this project will not be considered.

3

Background

Chapter 1 shortly introduced the basics of lazy and strict semantics. This chapter will present a more in-depth description of semantics and compilers, as well as some previous work on the subject.

3.1 Semantics

Every programming language has its own *semantics*. Semantics are often informally explained in English prose, but some languages have formal semantics. The semantics of a programming language defines the meaning of programs [7]. In other words, the semantics of a language describes how programs are executed. Different languages tend to use the same constructs, e.g. `if/else`, but evaluate these constructs differently. This is where the semantics defines the language.

The semantics of a language can be defined in different ways. The most natural approach for the semantics of this project is *operational semantics*. Fernández states that operational semantics is very useful for implementing languages and proving correctness of compiler optimisations [7]. By using operational semantics, each construct is defined by computation steps. These steps are often described with the help of transition systems, presenting the computation steps as a sequence of transitions. For this project, however, the operational semantics will be defined in the form of Haskell functions. By defining the operational semantics as Haskell functions, it can be observed and executed as Haskell code.

3.1.1 Strict semantics

Strict programming indicates that expressions are always fully evaluated when they appear in the chain of evaluation. For example, when an arithmetic expression is bound to a variable, the expression is fully evaluated to become a numerical value before being assigned to the variable. The value is then accessible by simply using the variable.

3.1.2 Lazy semantics

The main characteristics of laziness is delaying evaluation of expressions until they are needed. This is often done when binding expressions to variables. In order to delay the evalution of an expression, lazy semantics makes use of a data structure called *thunk* [8]. A thunk contains a suspended expression together with information that is required to finish the evaluation of the expression. Expressions that are to be evaluated are suspended in so called thunk values and passed on correspondingly. When needed, these thunks are *forced* in order to yield an actual usable value.

3.1.3 Pros and cons of both semantics

One reason to why applications are programmed strictly is because it can be necessary to have all values fully evaluated. If the evaluation of an expression that is erroneous is delayed, the error that it will bring will also be delayed. An example would be to consider a stateful system; if an operation is dependent on a reference, the value assigned to that reference needs to have been evaluated. If the evaluation of that value is delayed until after it is used, an error is prone to occur, as that value may not be there. By using strict evaluation, the evaluation is not delayed, avoiding any ambiguity of knowing whether or not references will have valid values.

It is not always certain that all expressions that are contained in a program will be used every time that it is executed. The more complex a program is, the higher the chance is for branching. For example, a number of expressions may be evaluated and bound to variables before a chain of `if`/`else if` expressions. Certain branches of this program may only use a select few, or even none, of the previously assigned variables. In such cases, it would not be optimal to actually evaluate them.

Lazy evalution can be motivated with an opposite reason in relation to strict evaluation. Programs that have a number of expressions bound to variables may not make use of all of them. For example, when using the `case` construct to pattern match in Haskell, the expression that is pattern matched can be assigned to a variable as an effect of the pattern matching:

```
case list of
  [] -> ...
  xs -> ...
```

In the second case, `list` is assigned to the variable `xs` as an effect of the pattern matching. For the pattern matching, `list` is evaluated to check its outermost constructor. If it is an empty list, it will be matched to the `[]` case, while the `xs` case handles all other cases, where `list` is bound to the variable `xs`. Further evaluation of `list` is delayed until `xs` is used, meaning that the actual content of list can be infinite in length without having the evaluation of the program ending up in a non-terminating loop.

Infinite data structures are a major benefit for using lazy evaluation. As described in Section 1.1, the function `take` in Haskell returns a subset of a given list. `take` has two arguments, an integer `x` and a list `xs`. This is used to get the `x` first elements of the

list `xs`. The lazy evaluation of `xs` in `take` allows the list to be infinite in size. This can be used for a program that e.g. returns the first `x` elements of the Fibonacci numbers.

3.2 Compilers

When a programmer has written code, the code must be translated into a form such that it can be executed by a computer. This translation is the job of a *compiler* [9]. A compiler is a program that reads programs in a *source* language and translates it to a *target* language. The source language is often some high-level language, while the target language is a low-level machine language. For example, when compiling programs with Java as source language, the target language that the compiler translates the code to is most likely Java Virtual Machine (JVM) [10].

There exists a certain category of compilers that translates code from a high-level language to another high-level language. This category of compilers is often referred to as *transpilers* or *source-to-source compilers* [11]. The goal of transpilers is to generate code in the target language that behaves equivalently to the code in the source language. The compiler defined in this thesis can be considered to be of this category. In this case, the source language is lazy CakeML and the target language is strict CakeML.

A common practice of compilers is to create an intermediate representation of the input code in the form of constructors in a syntax tree [9]. The intermediate representation needs to be easy to produce and is used for optimising the code, as well as allowing for easy generation of code in the target language. For this project, the intermediate representation of CakeML expressions is written as abstract syntax trees (ASTs) in Haskell. The ASTs will be altered so that when the target CakeML code is generated, it should be able to be evaluated lazily, but with the use of the native CakeML semantics that evaluates strictly.

3.3 CakeML

This section will present CakeML, the language that is the main focus of this thesis. CakeML is a strict functional programming language that is based on a subset of Standard ML [12]. Its semantics is defined in higher-order logic (with HOL interactive theorem prover) [6] and its compiler is formally proved to transform CakeML programs to semantically equivalent machine code [13].

The semantics provided for this project is specified in Lem [14, 15]. Lem is a lightweight tool for creating and handling large scale semantic definitions. It can also be used as an intermediate language for porting definitions between interactive theorem proving systems, such as HOL4.

This thesis presents a Haskell translation of the Lem definition of CakeML's strict semantics, as well as a newly defined lazy semantics of CakeML. The compiler that is the goal of the project draws inspiration from the lazy semantics.

3.4 Previous work

This section will present and discuss previous work that is related to this project. Topics include programming language libraries that allow lazy/strict evaluation in languages that do not allow it by default, as well as an evaluation strategy for Haskell that tries to optimise the inherent laziness of the language.

3.4.1 Libraries

There are many languages that are strict that have additional libraries containing lazy versions of primitive functions and vice versa. This section will present some examples of languages with this type of functionality.

Haskell

Haskell is one of very few languages that are inherently lazy. All primitive operations are lazy by nature, but the language also has certain functions that enforce strict evaluation.

An example of a function that enforces strictness is `seq` [16], which is a function that takes two arguments: `a` and `b`, and returns `b`. The result of `seq` is dependent on both `a` and `b`. If `a` does not terminate, `seq` does not terminate. If `a` does terminate, the result of `seq` is equal to `b`. While both `a` and `b` are evaluated before returning the result, `seq` does not guarantee that `a` is evaluated before `b`.

Haskell also has a function denoted as `$!` that evaluates arguments strictly. As arguments are normally evaluated lazily in Haskell, thunks are naturally generated, often creating an overhead in the computation. When an argument is known to always be evaluated, `$!` is recommended to avoid unnecessary suspensions [17].

C#

C# is an object oriented programming language developed by Microsoft. The language is strict, but has a Class `Lazy<T>` that allows lazy initialization [18]. The `Lazy<T>` class allows lazy evaluation in a similar way as it is done in Haskell, as function calls can be encapsulated for passing the call until it needs to be evaluated [19]. The call only happens once, with the use of caching, meaning that multiple uses will not cause multiple evaluations of the same suspended function.

3.4.2 Optimistic evaluation

This section will present and discuss an evaluation strategy that was developed for Haskell by Ennals and Peyton Jones [8]. In their paper they state that they measure common usage of thunks. As lazy evaluation creates an overhead with thunk utilization, programs can sometimes evaluate slower in a lazy semantics than in a strict semantics. This led them to create an evaluation strategy that makes use of their observation. Strict evaluation makes use of call-by-value mechanics, where expressions are always evaluated to values, while lazy evaluation uses call-by-need mechanics, where thunks are

exploited to delay the evaluation. The evaluation strategy presented in the paper finds expressions that will always be evaluated, applying call-by-value semantics on them, and uses an abortion mechanism for evaluation that takes too long.

The results of the work showed that the tested set of programs was sped up by a mean of about 15%, while no program was slowed down with more than 15%. Thus, by dynamically choosing between call-by-value and call-by-name, performance can be improved by a significant amount.

4

Strict semantics of CakeML

As stated in Section 2.1, the first major step of this project is to write lazy semantics for CakeML in Haskell. However, before this can be done, the existing strict semantics must be defined in Haskell. The definition includes the basic abstract syntax tree and semantic primitives of the language.

4.1 Definition

CakeML is a language with a semantics defined prior to this project. The compiler for this project is written in Haskell and produces CakeML syntax in the form of its Haskell representation. This means that the semantics must be defined in Haskell in order for it to be compatible with the code that the compiler produces. This is what this section will describe: defining the strict semantics of CakeML in Haskell.

The semantics of CakeML is defined in Lem [14], as described in Section 3.3. At the beginning of this project, a link to CakeML's GitHub repository was provided, containing the semantics in the form of *.lem* files. The goal of this part of the thesis is thus to manually translate the semantics from Lem to Haskell.

In order to translate the semantics of CakeML, three items must be investigated and manually transcribed from Lem to Haskell:

- The abstract syntax of CakeML
- Helper functions for CakeML's semantics
- The top-level interpreter function `evaluate`

All three items are defined as Lem modules in the provided GitHub repository and need to be defined as Haskell modules. Details about the contents and definition of these three items will be described in the following sections.

```
type exp =
  (* Literal *)
  | Lit of lit
  (* Variable *)
  | Var of id varN
  (* Anonymous function *)
  | Fun of varN * exp
  (* Application of operators to arguments *)
  | App of op * list exp
  (* Pattern matching *)
  | Mat of exp * list (pat * exp)
  (* Let expression that (potentially) binds a value to a variable
     in the environment and evaluates another expression *)
  | Let of maybe varN * exp * exp
  (* Local definition of (potentially) mutually recursive functions *)
  | Letrec of list (varN * varN * exp) * exp
  ...
```

Figure 4.1: Some expressions of CakeML defined in Lem.

```
(* Literal constants *)
type lit =
  | IntLit of integer
  | Char of char
  | StrLit of string
  ...
```

Figure 4.2: Some literals of CakeML defined in Lem.

4.1.1 Abstract syntax

The abstract syntax is the basic syntactical representation of the grammar of a language [20]. This includes the representation of literals, operations, and expressions. As Lem and Haskell are fairly similar in structure, translating the abstract syntax from Lem to Haskell is not an all too difficult task. With the use of algebraic datatypes, each construct in the grammar is defined as its own type. For example, some of the expressions of CakeML are represented in Lem in Figure 4.1. The constructors are complemented with data from other datatypes, e.g. `lit` and `op`, that are also defined as a part of the language. These datatypes are shown in Figures 4.2 to 4.5.

The Lem definition of CakeML expressions in Figure 4.1 is translated to Haskell in Figure 4.6. Similarly, complementing datatypes, such as `lit` and `op`, as well as other constructs in the grammar, are also translated as needed. These datatypes can be seen in Figure 4.7.

```
(* Operators *)
type op =
  (* Operations on integers *)
  | Opn of opn
  | Opb of opb
  (* Function application *)
  | Opapp
  (* Reference operations *)
  | Opassign
  | Opref
  | Opderef
  ...


(* Arithmetic operators *)
type opn = Plus | Minus | Times | Divide | Modulo

(* Comparison operators *)
type opb = Lt | Gt | Leq | Geq
```

Figure 4.3: Some operators of CakeML defined in Lem.

```
(* Identifiers *)
type id 'a =
  | Short of 'a
  | Long of modN * 'a


(* Variable names *)
type varN = string

(* Module names *)
type modN = string

(* Constructor names *)
type conN = string
```

Figure 4.4: Identifiers and some names of CakeML defined in Lem.

```
(* Patterns *)
type pat =
  | Pvar of varN
  | Plit of lit
  | Pcon of maybe (id conN) * list pat
  ...
```

Figure 4.5: Some patterns of CakeML defined in Lem.

```
data Exp
  -- Literal
  = Literal Lit
  -- Variable
  | Var (Id VarN)
  -- Anonymous function
  | Fun VarN Exp
  -- Application of operators on arguments
  | App Op [Exp]
  -- Pattern matching
  | Mat Exp [(Pat, Exp)]
  -- Let expression that (potentially) binds a value to a variable
  -- in the environment and evaluates another expression
  | Let (Maybe VarN) Exp Exp
  -- Local definition of (potentially) mutually recursive functions
  | LetRec [(VarN, VarN, Exp)] Exp
  ...
```

Figure 4.6: Haskell definition of some CakeML's expressions.

These expressions are the basic building blocks that a programmer uses to create their programs. For example, a simple integer such as 5 is represented as

```
Literal (IntLit 5)
```

when using this abstract syntax. Building more complex expressions is as simple as combining multiple expressions. For example, with the `Fun` expression, it is possible to create functions that take arguments and produces values. For example, `take` that is described in Section 1.1 can be defined by combining a number of expressions. The definition of `take` using CakeML expressions in Haskell can be seen in Figure 4.8.

4.1.2 Helper functions for CakeML's semantics

CakeML's semantics has many helper functions that define how computations are performed. This includes operations such as basic arithmetic, logical operations, lookups, and pattern matching. Also included in the semantic primitives are the inner representations of *values*, *state*, and *environment*.

```
-- Literal constants
data Lit
  = IntLit Int
  | Char Char
  | StrLit String
  ...


-- Operators
data Op
  -- Integer operations
  = OPN Opn
  | OPB Opb
  -- Function application
  | OpApp
  -- Reference operations
  | OpAssign
  | OpRef
  | OpDeref
  ...

-- Arithmetics operators
data Opn = Plus | Minus | Times | Divide | Modulo

-- Comparison operators
data Opb = Lt | Gt | LEq | GEq


-- Identifiers
data Id a = Short a | Long ModN a


-- | Patterns
data Pat
  = PVar VarN
  | PLit Lit
  | PCon (Maybe (Id ConN)) [Pat]
  ...
```

Figure 4.7: Haskell definitions of some complementing CakeML datatypes.

```
cakeTake =
  LetRec [("take", "n",
           Fun "ls" $
           If (App Equality [Var (Short "n"), Literal (IntLit 0)])
            (Con (Just (Short "nil")) [])
            (Mat (Var (Short "ls"))
             [(PCon (Just (Short "::" )) [PVar "elem", PVar "rest"]
              ,Con (Just (Short "::")) [Var (Short "elem")
                                       ,cakeTake2])
             ,(PCon (Just (Short "nil")) []
              ,Con (Just (Short "nil")) [])]
            )
          )] $
  Var (Short "take")
  where cakeTake2 =
          App OpApp [App OpApp [Var (Short "take")
                               ,decr]
                     ,Var (Short "rest")]
        decr =
          App (OPN Minus) [Var (Short "n")
                          ,Literal (IntLit 1)]
```

Figure 4.8: The recursive function `take` defined by combining CakeML expressions in Haskell.

```
type v =
  (* Literal values *)
  | Litv of lit
  (* Function closures *)
  | Closure of environment v * varN * exp
  (* Function closure for recursive functions *)
  | Recclosure of environment v * list (varN * varN * exp) * varN
  ...
```

Figure 4.9: Some values of CakeML defined in Lem.

Values are represented by the datatype `v` (in Haskell it is denoted with an upper case `V`) and are what expressions usually are evaluated to. Some of the values of CakeML are represented in Lem as seen in Figure 4.9.

In order, these values represent literals, closures, and recursive closures. Just as with the expressions described above, other datatypes that represent other parts of the grammar are needed. However, these mostly already exist from the definition of the abstract syntax and are therefore reused.

The translation of these values to Haskell is performed analogously to the abstract

```
data V
  -- Literal values
  = LitV Lit
  -- Function Closures
  | Closure (Environment V) VarN Exp
  -- Function closure for recursive functions
  | RecClosure (Environment V) [(VarN, VarN, Exp)] VarN
  ...
```

Figure 4.10: Some values of CakeML defined in Haskell.

syntax described in Section 4.1.1 as seen in Figure 4.10.

An example of an expression and its value counterpart is the expression $1 + 2$, whose result is the value 3. The expression is represented as

```
App (OPN Plus) [Literal (IntLit 1), Literal (IntLit 2)]
```

which is evaluated to the value

```
LitV (IntLit 3)
```

The helper function used to actually apply the addition to the literals is `do_app`, which pattern matches on the given operation and arguments and uses another helper function `opn_lookup` to find the `(+)` operator:

```
do_app s op vs =
  case (op, vs) of
    (OPN op, [LitV (IntLit n1), LitV (IntLit n2)]) ->
      Just (s, RVal (LitV (IntLit (opn_lookup op n1 n2))))
...

opn_lookup n =
  case n of
    Plus   -> (+)
...
```

4.1.3 The top level interpreter function `evaluate`

Linking the abstract syntax to the semantics of CakeML is the `evaluate` function. The job of `evaluate` is to take a list of expressions as input and return a state together with a result. A result can consist of a list of values or an error. Also contributing to the evaluation of expressions are a state and an environment, which are given as input to `evaluate`. The type of `evaluate` (in Lem notation) is thus:

```
evaluate:state -> environment v -> list exp -> state*result (list v) v
```

Expressions are taken either as a list in `evaluate`, in which case they are evaluated sequentially, or as singletons, in which case they are pattern matched to be evaluated with unique strategies:

```
let rec
evaluate st env []           = (st, Rval [])
and
evaluate st env (e1::e2::es) = ...
and
evaluate st env [Lit l]      = (st, RVal [Litv l])
and
...
```

The logic of each case mostly consists of applying appropriate operations from the semantic helper functions. For example, when evaluating the `Var n` expression, the `lookup_var_id` helper function is applied to `n` in order to check if a value is assigned to the variable in the environment. In the case it does exist, the value is returned:

```
evaluate st env [Var n] =
  match lookup_var_id n env with
  | Just v -> (st, Rval [v])
  | Nothing -> (st, Rerr (Rabort Rtype_error))
  end
```

Expressions that contain sub-expressions (e.g. binary arithmetic operations) often require that one or more of these sub-expressions are evaluated to values in order to use them as input for the semantic helper functions. For example, the evaluation of the expression `If e1 e2 e3` requires that `e1`, which is the condition, is evaluated first in order to use the semantic function `do_if` to decide whether `e2` or `e3` should be evaluated.

```
evaluate st env [If e1 e2 e3] =
  match st (evaluate st env [e1]) with
  | (st', Rval v) ->
      match do_if (head v) e2 e3 with
      | Just e -> evaluate st' env [e]
      | Nothing -> (st', Rerr (Rabort Rtype_error))
      end
  | res -> res
  end
```

The translation to Haskell maintains the same structure of `evaluate`. The type declaration for `evaluate` is transcribed to Haskell as:

```
evaluate :: State -> Environment V -> [Exp] -> (State, Result [V] V)
```

Lists and pattern matching are used in similar ways in the Haskell definition:

```
evaluate st env []          = (st, RVal [])
evaluate st env (e1:e2:es)  = ...
evaluate st env [Literal l] = (st, RVal [LitV l])
```

The logic of `evaluate` is also maintained in the translation to Haskell. For example, the `Var n` expression is defined in Haskell as follows:

```
evaluate st env [Var n] =
  case lookup_var_id n env of
    Just v  -> (st, RVal [v])
    Nothing -> (st, RErr (RAbort RType_Error))
```

4.2 Example of evaluation

By using `evaluate`, CakeML expressions are evaluated to values. This section will show how such an expression is evaluated by using the definition of `take` described in Section 1.1 and shown in abstract syntax at the end of Section 4.1.1.

When `take` is evaluated, the first grammatical construct that is encountered is `LetRec` that creates a `RecClosure` when evaluated. When the `RecClosure` is applied to an integer literal with the use of the `App OpApp` expression, the literal will be bound to the variable `n`. The expression representing `take` in the `RecClosure` is a `Fun`, which is evaluated to a `Closure` that takes a list `ls`. When the `Closure` is applied to a list with `App OpApp`, the actual logic that is described in Section 1.1 will be carried out. When the `If` expression is evaluated, the condition is evaluated first: `n` is compared with 0 to check if they are equal. If they are equal, the constructor `nil` is returned by the `do_if` helper function, which will be evaluated to a constructor value. If `n` is not equal to 0, the second expression is returned by `do_if`, which pattern matches on the list `ls` with the `Mat` expression to check if it is either a `::` or `nil` constructor. As `ls` is evaluated for this check, it will be fully evaluated by the evaluation steps of the `Con` expression. In the case of `::`, the head element is stored as the variable `elem`, and the tail of the list is stored as the variable `rest`. Then a `Con` expression for a list is created with `elem` as its head element and the application of `take` on `n` subtracted by 1 and the tail of the list `rest`. In the case of `nil` being the constructor of `ls`, a `nil` constructor is simply returned, ending the computation.

4.3 Testing the definition

To make sure that CakeML's semantics are properly translated to Haskell, testing is performed. Tests consist of running `evaluate` on a number of predefined expressions together with dummy instances of state and environment. The results of the evaluations are then compared with expected values in the form of unit testing to check for any deviations.

Faulty results indicate erroneous implementation of the semantics. Through repeated troubleshooting and amending code, the semantics translation has reached a satisfactory state that evaluate expressions correctly, allowing the next phase of the project to begin: defining the lazy semantics.

5

Lazy semantics of CakeML

The second step of this thesis, after translating the strict semantics of CakeML to Haskell, is to define lazy semantics that the compiler will be based on. The lazy semantics defined in Haskell is designed to delay evaluation by storing partially evaluated expressions in thunks. This chapter will describe the definition of the lazy semantics.

5.1 Definition

The task of defining the lazy semantics for CakeML consists of defining a new `evaluate` function (called `evaluateLazy` from now on) and appropriate functions in the semantic primitives. In order to implement laziness, it is also necessary to incorporate the idea of thunks, as described in Section 3.1.2. As such, thunks are added as the `Thunk` constructor in the `V` datatype. This makes it a value that can be returned as a result by `evaluateLazy`. The main purpose of using thunks is to suspend partially evaluated expressions within. To further evaluate suspended expressions, additional information, contained in the environment, is required. The `Thunk` constructor is thus added to the `V` type:

```
data V =
  LitV Lit
  ...
  | Thunk (Environment V) Exp
```

The function `evaluateLazy`, the lazy version of `evaluate`, is implemented with the goal of evaluating expressions as little as possible. While the type of `evaluateLazy` needs to stay similar to `evaluate`, minor changes are made; mainly, the state is omitted altogether in order to keep the evaluation pure. The type of `evaluateLazy` is thus declared as follows:

```
evaluateLazy :: Environment V -> [Exp] -> Result [V] V
```

With the addition of the `Thunk` constructor, `evaluateLazy` returns either a thunk or any other value.

Another vital function that needs to be implemented is `force`. This function is designed to take a value and pattern match for the `Thunk` constructor. When the `Thunk` case is entered, the expression wrapped inside the thunk is extracted and lazily evaluated until an actual value is yielded. When any other value is caught in the pattern matching, the value is simply returned as a result. When the evaluation of an expression ends at top level, the returned value must be an actual non-thunk value, as in the strict semantics of CakeML. The function `force` is defined as such:

```
force :: V -> Result [V] V
force (Thunk env e) =
  case evaluateLazy env [e] of
    RVal [Thunk env' e'] -> force (Thunk env' e')
    res -> res
force v = RVal [v]
```

Using the definition of thunks and `force`, the definition of `evaluateLazy` can be written. The following sections will describe how it is defined in two attempts, one that is erroneous and one that is correct.

5.1.1 Attempt 1

The first (erroneous) attempt of implementing `evaluateLazy` uses a naive idea of evaluating expressions as little as possible. This is mainly seen in expressions that contains sub-expressions, e.g. `If e1 e2 e3`. The first version of `evaluateLazy` on `If` applies `evaluateLazy` on `e1` and checks if it returns a thunk or a value. If it is a thunk, wrapped together with a partially evaluated expresion `e1'` and a potentially different environment `env'`, it will create and return a new thunk `Thunk env' (If e1' e2 e3)`. If any other type of value `v` is returned, it applies `do_if` to yield either `e2` or `e3` as the expression `e`. This in turn, assuming that `v` is valid, creates and returns a new thunk containing `e` and the environment that was given as input to `evaluateLazy`.

```
evaluateLazy st env [If e1 e2 e3]   =
  case evaluateLazy st env [e1] of
    (st', RVal vs)   -> case head vs of
      Thunk env' e1' -> (st', RVal [Thunk env' (If e1' e2 e3)])
      v              -> case do_if v e2 e3 of
        Just e  -> (st', RVal [Thunk env e])
        Nothing -> (st', RErr (RAbort RType_Error))
    res -> res
```

The problem with this implementation is that the environment is passed along in each iteration. The strict `evaluate` evaluates `e` with the input environment. This act of passing on an altered environment is an unwanted process, as `e1` may assign some variables that were only intended to be temporary for the evaluation of `e1`. The strict

`evaluate` does not exhibit this behaviour, but instead keeps the evaluation of `e1` in its own scope.

5.1.2 Attempt 2

The second attempt of implementing `evaluateLazy` creates a delayed evaluation without altering more data than wanted. This is achieved by forcing certain sub-expressions instead of leaving them suspended. Forcing these sub-expressions produces the value that is needed to continue the evaluation process. The forcing process is made easier by creating a function called `evalAndForce` that applies `evaluateLazy` on the head of the given list of expressions and environment, and forces the resulting value.

```
evalAndForce :: Environment V -> [Exp] -> Result [V] V
evalAndForce _env []    = RVal []
evalAndForce env (e:es) =
  case evaluateLazy env [e] of
    RVal v -> case force (head v) of
      RVal val ->
        case evalAndForce env es of
          RVal vs -> RVal ((head val):vs)
          res -> res
      res -> res
    res -> res
```

Taking the `If e1 e2 e3` case as an example again, the condition `e1` is forced to produce a (hopefully boolean) value, which is run through `do_if`. The resulting expression, which is the branch that is to be evaluated in the next step, is wrapped in a thunk and returned as a result.

```
evaluateLazy env [If e1 e2 e3]   =
  case evalAndForce env [e1] of
    RVal v ->
      case do_if (head v) e2 e3 of
        Just e  -> RVal [Thunk env e]
        Nothing -> RErr (RAbort RType_Error)
    res -> res
```

This new idea of forcing sub-expressions is applied to all cases of the pattern matching in `evaluateLazy`.

5.2 Example of evaluation

Similarly to Section 4.2, this section will describe how the lazy evaluation of `take` works. The evaluation described will be using `force` so that the top-level result will not be a thunk value.

The first step of evaluating `take` is the evaluation of the `LetRec` expression. It will create a `RecClosure` and return the following expression (using the variable `take`) as a

thunk. When forced, it can be used in the same manner as with the strict semantics: applying the `RecClosure` that is bound to the variable `take` to an integer literal with `App OpApp`. The resulting `Fun` expression is evaluated to a `Closure` similarly to the strict semantics. When applied to a list `ls` with `App OpApp`, the `If` expression is evaluated. The expression that represents the boolean condition (equality of `n` and 0) is forced to yield a non-thunk value, which is used as an argument to the `do_if` helper function. The result when `n` is equal to 0 is the expression that returns a `nil` constructor. In the other case, the list `ls` is pattern matched with `Mat` to check if it is either a `::` or `nil` constructor. When `ls` is evaluated here, both the head element and the tail of the list (applied as argument to `take` with `n` subtracted by 1) are put into thunks. This means that forcing constructors by normal means will not yield a fully evaluated constructor value. Instead, a different `force` function is defined to specifically force the contents of constructors. This is done by simply pattern matching on constructor values to check for thunks that are then evaluated and forced to yield a fully evaluated list.

5.3 Testing the definition

Similarly to the strict semantics, unit testing is performed on the lazy semantics by using the same dummy environment and expressions. The result from the lazy evaluation is compared to what the strict evaluation yields to make sure that they are equivalent. An additional test method is applied for lazy evaluation as well. This testing consists of checking for termination in evaluation of expressions that normally do not terminate with the strict semantics. These two test methods will be described in this section.

5.3.1 Comparing results

For comparing results between lazy and strict evaluation, a function, called `compareEval` is defined. The function `compareEval` takes a list of expressions and applies `evaluate` and `evaluateLazy` in separate passes. Both evaluation processes use the same dummy environment. Evaluation with an empty state (called `empty_st`) and a dummy environment (called `ex_env`) is simplified by defining a function for strict evaluation:

```
ex e = evaluate empty_st ex_env [e]
```

Similarly, evaluation for lazy evaluation with `ex_env` is simplified by defining a function for lazy evaluation:

```
exForce e = evalAndForce ex_env [e]
```

`evalAndForce` is naturally used, as a pure value is expected to be compared with the strict evaluation. With the definition of `ex` and `exForce`, `compareEval` is defined as such:

```
compareEval expList = map exForce expList == map (snd . ex) expList
```

By collecting a list of expressions, testing is performed by simply running `compareEval` on a given list. This is done whenever the semantics is changed and needs to be tested to control that the results would not change.

One expression that is not compared is the `Con` expression. The lazy evaluation of `Con` expressions produces thunks of the constructors' arguments. When forced, they are not evaluated. This behaviour allows the possibility of infinite data structures, such as lists, to exist. Keeping the contents of a list as thunks can save computation time if, for example, `take` is applied to a significantly large list, but only takes a minimal amount of elements. However, thunks can also act as a double-edged sword. For example, in the case where a full list is to be shown, the strict semantics will be faster to generate the list. This is because of the additional steps that the lazy semantics takes to produce thunks only to force them to non-thunk values, which the strict semantics produces directly. This example shows that lazy semantics is not always optimal and that different situations may require different semantics for the best performance.

5.3.2 Testing termination

As laziness brings the feature of delaying evaluation of expressions, certain expressions that would lead to non-termination when using the strict semantics should terminate when using the lazy semantics. For example, when the expression `Let x e1 e2` has a non-terminating expression for `e1`, the strict semantics will not terminate on the evaluation of `e1`. However, the lazy semantics will simply wrap `e1` in a thunk and bind it to the variable `x`, allowing the evaluation to continue to evaluating `e2`.

This test method consists of creating CakeML expressions that exhibit the behaviour explained above. One such expression is defined and called `termExp`. A non-terminating expression `inf` is also defined to be used together with `termExp`. The expression `inf` is defined to create a recursive function `fun` that simply calls itself to create an infinite loop. The definition of `termExp` uses `inf` by binding it to a variable `var` in a `Let` expression before returning a string literal "OK".

```
termExp =
  Let (Just "var") (inf) (Literal (StrLit "OK"))
  where inf =
    LetRec [("fun", "par", App OpApp [Var (Short "fun")
                                     ,Literal (IntLit 0)])]
    (App OpApp [Var (Short "fun"), Literal (IntLit 0)])
```

Running `termExp` with the strict semantics naturally causes a non-terminating loop, while running `termExp` with the lazy semantics results in the literal value "OK" being returned, as expected.

With the test results showing positive results, the lazy semantics can be deemed to work as intended. This signals that the groundwork for the compiler is complete.

6

Lazy to strict compilation

The third major step of this project is to implement the compiler that takes lazy CakeML expressions and produces CakeML expressions that retain the lazy behaviour even when evaluated using the strict semantics. The compiler is implemented in Haskell by using the ideas behind the definition of the lazy semantics. This chapter will describe the implementation of the compiler that compiles CakeML expressions from the lazy semantics to the strict semantics in such a way that the lazy behaviour is preserved.

6.1 Implementation

For the task of creating the compiler, a `compile` function is defined. The `compile` function is designed to take an expression and alter it to another expression that incorporates the logic of thunks and `force` defined in Section 5.1. In order to further explain how this is done, more information about how thunks and `force` are defined needs to be detailed.

6.1.1 Thunks as defined in the compiler

In the lazy semantics, thunks are defined as values. As the compiler only makes use of expressions, values are not available as resources. The idea is to simulate the behaviour of storing partially evaluated expressions wrapped inside thunks in the environment as done in the lazy semantics. For this purpose, it is necessary to store an expression together with the environment that is in use when the expression is originally supposed to be evaluated.

Thunks are defined with the use of the `Fun` expression combined with the `Con` expression. When evaluated, `Fun` expressions become closures that contain the input expression together with the environment in use at the time of evaluation. This fulfills the requirements of keeping the expression together with an environment.

The `Con` expressions are constructors that allow separation between thunks and values. Thunks and values are separated with constructor names that uniquely identify

them. For an expression `e`, thunks are defined and simplified with the Haskell function `makeThunk`:

```
makeThunk e = Con (Just (Short "Thunk")) [Fun "" e]
```

As `e` is put in a `Fun` expression, it is not actually evaluated. Instead, it is put inside the resulting closure, awaiting evaluation when used as an argument to an `App Op` expression.

A value in the case of the compiler is an expression that does not require a delayed evaluation. This mainly applies to expressions such as `Lit`, `Fun`, and `Con` that do not require any further evaluation other than returning the appropriate value. When wrapping a value around a `Literal` expression, there is no need to keep the environment with it, as the evaluation is not to be delayed. Thus, for an expression `e`, values are defined and simplified with the Haskell function `makeVal`:

```
makeVal e = Con (Just (Short "Val")) [e]
```

In order for the constructors `Thunk` and `Val` to be usable, they need to be defined under a type identifier in the environment. Thus, during the testing phase for the compiler, the constructors are defined under the type identifier `lazy`.

6.1.2 Defining `force` in the compiler

With the definition of thunks and values finished, the task of defining `force` remains. In the lazy semantics, `force` pattern matches on values to recursively evaluate expressions inside of the input thunks until they stop producing thunks.

In order to simulate the behaviour of `force` defined for the lazy semantics, a number of CakeML expressions are combined: `LetRec` for recursion, `Mat` for pattern matching, and `App OpApp` both for applying `force` recursively and forcing an evaluation of the closure contained in a thunk.

```
force :: Exp -> Exp
force e =
  App OpApp [LetRec [("force", "exp"
                     , Mat (Var (Short "exp"))
                       [(PCon (Just (Short "Thunk")) [PVar "Thunk"]
                        , App OpApp [Var (Short "force")
                                     , App OpApp [Var (Short "Thunk")
                                                  , Literal (IntLit 0)]])
                       ,(PCon (Just (Short "Val")) [PVar "Val"]
                        , Var (Short "Val"))]
                     )] (Var (Short "force"))
             , e]
```

This definition of `force` is based on the Haskell definition seen in Section 5.1. The abstract syntax shown above is used by the compiler. The concrete syntax for `force`, which is easier for the human eye to read, can be defined as such:

```
fun force e =
  case e of
    thunk t => force (t 0)
  | val v   => v
```

By injecting `force` into the code, actual values are generated from thunks. When a `lazy` constructor is given as input to `force`, it is pattern matched to check if it is a `Thunk` or a `Val`. Similarly to the definition of `force` for the lazy semantics in Section 5.1, when a `Thunk` constructor is given as input, the content of the constuctor is extracted and forced. In the case of the `Val` constructor, the content is simply extracted and returned.

6.1.3 The `compile` function

With the important elements of thunks and `force` implemented, the next step is to utilise them in the `compile` function. As `compile` takes an expression and returns an expression, the type of the function is

```
compile :: Exp -> Exp
```

Pattern matching for all expressions in Haskell allows the function to handle each expression uniquely. With the lazy semantics defined in Chapter 5 in mind, `compile` is implemented with a similar approach. Expressions are wrapped in either `Thunk` or `Val` constructors. When an expression needs to be fully evaluated, `force` is applied.

An example of compiling an expression can be described with the expression `If e1 e2 e3`. As seen in Section 5.1.2, `e1` needs to be forced, while `e2` or `e3` should be returned as a thunk. This is handled by applying `force` on `e1` after compiling it, as well as wrapping both `e2` and `e3` in thunks after compiling them. The code is simplified by creating helper functions `forceCompile`, which is a function composition of `force` and `compile`, and `thunkCompile`, which is a function composition of thunk wrapping and `compile`.

```
If (forceCompile e1) (thunkCompile e2) (thunkCompile e3)
```

When evaluated, this expression takes the following steps:

1. `e1` is forced and evaluated to the value `v`.
2. `v` is used as argument to `do_if` together with `e2` and `e3`, both compiled and wrapped in thunks.
3. The resulting expression `e` is evaluated.

In order for the evaluation of the compiled `If` expression to yield the same value that the uncompiled version would yield, it simply needs to be forced before being evaluated.

This logic of wrapping expressions in `Thunk` and `Val` constructors is applied to all cases of `compile`. Testing (described in Section 6.3) showed that all expressions evaluate correctly, except for one case: the `LetRec` expression. This issue (and its solution) will be explained in the following subsection.

6.1.4 Compiling `LetRec`

The expression `LetRec` is what makes recursive definitions possible. The expression consists of a list of local function definitions that each have a function name, argument name, and a function body. This list is followed by another expression to be evaluated. When a `LetRec` is evaluated, its function definitions are stored in the environment as `RecClosure` values, each containing environments that contain all of the function definitions.

Evaluation of `LetRec` expressions is defined (in Haskell) as such:

```
evaluate st env [LetRec funs e] =
  if allDistinct (map (\(x,y,z) -> x) funs) then
    evaluate st (env {v = build_rec_env funs env (v env)}) [e]
  else
    (st, RErr (RAbort RType_Error))
```

The problem with the compiler and this evaluation strategy is that it is the helper function `build_rec_env` that creates the `RecClosure` values and stores them in the environment. This means that the compiler cannot wrap the `RecClosure` values in the `Val` constructor, as the semantics is not a part of the compiler. `RecClosure` values are thus naked in the environment, causing `force` to result in a type error when called, as it expects the constructors `Thunk` and `Val` in the pattern matching.

In order to create valid `RecClosure` values in the environment, a workaround is needed. This workaround consists of allowing the `evaluate` function to create the `RecClosure` values that are incompatible with `force` and redefine them as valid values. All this is done in the definition of `compile` for `LetRec`:

```
compile (LetRec funs e) =
  LetRec (recVals funs) (repl (compile e) (fst3 funs))
  where
    fst3 []           = []
    fst3 ((f,_,_):fs) = (f:fst3 fs)
    repl e []     = e
    repl e (f:fs) = Let (Just f) (makeVal (Var (Short f))) (repl e fs)
    recVals []           = []
    recVals ((f,x,e):xs) =
      (f,x,
       Let (Just f)
       (makeVal (Var (Short f)))
       (compile e)
      ):(recVals xs)
```

The helper functions are defined as such: `fst3` takes a list of tuples of three and creates a list of the first elements of the tuples (this is used to extract the function identifiers), `repl` creates new definitions of the `RecClosure` values by extracting them from the environment and storing them after wrapping them with a `Val` constructor, `recVals` goes into each function and creates a `Val` constructor for their respective expressions. The identifiers created in the first step (that creates naked `RecClosure` values) are

overshadowed in the environment when they are redefined. The lookup function for variables searches the variable list in the environment starting from the head and moves towards the tail, meaning that there will not be any problems with the wrong `RecClosure` being used.

6.2 Example of evaluation

Evaluating a compiled expression should yield an equivalent result to when evaluated with the lazy semantics. Similarly to Sections 4.2 and 5.2, this section will describe how `take` is evaluated after being compiled. The compiled expression is assumed to be forced at top level before being evaluated.

The evaluation of the compiled `LetRec` will yield a `RecClosure`. With the workaround described in Section 6.1.4, the `RecClosure` will be properly wrapped with a `Val` constructor for proper forcing. When applied with an integer literal, the evaluation of the `RecClosure` will return an evaluated `Fun` expression as a `Closure` wrapped in a `Val` constructor. When forced, the `Closure` can be applied to a list `ls` by using `App OpApp`. For the evaluation of the compiled `If` expression, as the condition is forced, it will yield a usable value. The two branches are wrapped in thunks, meaning that the result of `do_if` will yield a thunk. In the case where the condition is true (`n` is equal to 0), a `nil` constructor is simply returned. In the case where the condition is false, the list `ls` is pattern matched to check its constructor. While the compilation of the `Mat` expression will apply a force on `ls`, the evaluation of `Con` does not force the arguments, as described in Section 5.2. Thus, a new definition of `force` is needed, similar to the one defined for the lazy semantics. This separate `force` is used after evaluation to yield a fully evaluated list.

6.3 Testing the compiler

The compiler is tested similarly to the lazy semantics. Firstly, a number of common expressions are compiled and evaluated, followed by a comparison with the evaluation of the non-compiled expressions. Secondly, termination is tested to see if expressions that do not terminate when evaluated with the strict semantics do terminate after compiling and evaluating them with the same strict semantics.

The evaluation of non-compiled expressions uses the function `ex`, defined in Section 5.3.1, that uses an empty state and dummy environment, as when testing the lazy semantics. Evaluation of compiled expressions is simplified with a function called `efc` that compiles, forces, and evaluates a given expression:

```
efc = ex . force . compile
```

The two evaluations `ex` and `efc` are then compared with a function called `compareEval`:

```
compareEval :: Exp -> Bool
compareEval e = strict == lazy
  where strict = ex e
        lazy   = efc e
```

With the results of `compareEval` giving a `True` value when tested with a number of expressions, the compiler is deemed to work as intended.

Termination is tested for the compiler by running `efc` for expressions that do not terminate when simply evaluated with the strict semantics, but do terminate with the lazy semantics. For example, the expression defined for the testing of the lazy semantics:

```
termExp =
  Let (Just "var") (inf) (Literal (StrLit "OK"))
  where inf =
    LetRec [("fun", "par", App OpApp [Var (Short "fun")
                                     ,Literal (IntLit 0)])]
    (App OpApp [Var (Short "fun"), Literal (IntLit 0)])
```

When executed with `efc`, this expression gives the literal value ”OK”, which indicates that the compiler indeed produces expressions that exhibit laziness.

While the compiler at this stage successfully delays expressions in thunks until they are called, it is still not quite producing truly lazy expressions. True laziness is mostly associated with call-by-need semantics, where thunks retain the values that their expressions are evaluated to [20]. The semantics that the compiler is exhibiting at this stage is called *call-by-name*, where evaluations are simply delayed. If the thunk is used more than once, it will be evaluated each time it is used [9]. Call-by-need semantics can be seen as more beneficial than call-by-name, as repeated use of e.g. variables would require repeated use of thunks. The next stage of this project is thus to optimise the compiler by implementing call-by-need semantics.

7

Optimization: call-by-need semantics

With the completion of the compiler, expressions bound to variables are stored as thunks, delaying their evaluation until they are called. While this means the compiled code exhibits lazy behaviour, the thunks are evaluated every time that the variable is called. The optimal behaviour, called *call-by-need*, is to evaluate the thunks at most once and reuse the resulting value. This chapter will describe how the compiler is optimised to use the stateful features of CakeML in order to implement call-by-need.

7.1 Implementation

With the implementation of the compiler, as described in Chapter 6, expressions bound to variables are stored in the environment as thunks and evaluated when called. In order to use call-by-need mechanics, the compiler is altered to make use of CakeML's inherent state in order to save thunks, as well as their evaluated value.

Thunks are changed so that instead of containing delayed expressions, they contain references that point to the location of the delayed expressions in the state. Two new constructors are introduced under a new type to represent expressions and values stored in the state: `RefExp` and `RefVal`. Delayed expressions are wrapped in `Fun` expressions inside the `RefExp` constructor, similarly to how they were with the `Thunk` constructor before the optimisation. Fully evaluated values are wrapped in the `RefVal` constructor without any further modifications, just like with the `Val` constructor.

In order to be able to utilise the state, there are three operations that are vital: `OpRef`, `OpDeref`, and `OpAssign`. These are all defined under the `Op` datatype of the CakeML grammar, which is used in combination with the `App` expression. When evaluated, `App OpRef [e]` creates a new entry in the state for the value that `e` is evaluated to. The reference is then returned as a location value `Loc n`. `App OpDeref [e]` takes

an expression `e` that is to be evaluated to a reference (e.g. a variable that contains the reference), and returns the value that is stored in the state for the given reference. `App OpAssign [e1, e2]` takes a reference `e1` and an expression `e2`, and changes the value stored in the state in the location given by `e1` to the value that `e2` is evaluated to. The result of evaluating `App OpAssign` is a constructor expression containing `Nothing`, which needs to be kept in mind when using `OpAssign`.

By using the operations `OpRef`, `OpDeref`, and `OpAssign`, two operations in the compiler are changed. The first change of the compiler is made to how thunks are generated. This is done in the `makeThunk` function, which previously took an expression `e` and wrapped it in the `Thunk` constructor and `Fun` expression. With the optimization, `makeThunk` takes `e` and creates a reference in the state for a `Fun` expression wrapped around a `RefExp` constructor, and finally wraps the reference with a `Thunk` constructor:

```
makeThunk e =
  Con (Just (Short "Thunk"))
    [App OpRef
      [Con (Just (Short "RefExp"))
        [Fun "" e]
      ]
    ]
```

When evaluated, this expression adds the delayed expression to the state and returns a reference to its location. This means that each time a thunk is created, its expression will be stored in the state instead of simply being wrapped in a `Thunk` constructor and a closure. Similarly, values in the state are created with a function called `makeRefVal`:

```
makeRefVal e = Con (Just (Short "RefVal")) [e]
```

The second change made with the optimisation is how `force` behaves. As the `Thunk` constructor no longer contains a closure, but instead a reference, the `Thunk` case of the pattern matching in `force` uses the `OpDeref` operation to get what is stored in the state for the given reference. The reference can point at either a `RefExp` or a `RefVal` constructor, and must thus be pattern matched. When a `RefVal` is caught in the pattern matching, the contained value is simply returned as it is. For the `RefExp` case, the contained value is a delayed expression in a closure and must therefore be forced. The `RefExp` that the reference is pointing to is then changed with an `OpAssign` operation to a `RefVal` of the value that the `force` yielded. After the assignment, the evaluated value is returned. The new definition of `force` can be seen in Figure 7.1. The concrete syntax of the new definition of `force` can be seen in Figure 7.2.

With these re-definitions, the `compile` function does not need any changes and can be used as before the optimisation.

7.2 Example of evaluation

Evaluation of expressions is optimized after the implementation of call-by-need. The example described in Section 6.2 shows how `take` is evaluated after being compiled with call-by-name. The evaluation of `take` with call-by-need semantics differs only slightly,

```
force :: Exp -> Exp
force e =
  App OpApp [LetRec [("force", "exp"
                     , Mat (Var (Short "exp"))
                       [(PCon (Just (Short "Thunk")) [PVar "TPtr"]
                        ,refMat (App OpDeref [Var (Short "TPtr")]))
                       ,(PCon (Just (Short "Val")) [PVar "Val"]
                        , Var (Short "Val"))]
                     )] (Var (Short "force"))
             , e]

refMat :: Exp -> Exp
refMat e =
  Mat (e)
  [(PCon (Just (Short "RefVal")) [PVar "RefV"]
   ,Var (Short "RefV"))
  ,(PCon (Just (Short "RefExp")) [PVar "RefE"]
   ,Let Nothing (App OpAssign [Var (Short "TPtr"), makeRefVal sol])
     (getVal (App OpDeref [Var (Short "TPtr")])))
  ]
  where sol = App OpApp [Var (Short "force"),
                         App OpApp [Var (Short "RefE")
                                   ,Literal (IntLit 0)]]
        getVal e' = Mat e'
                    [(refValPat [PVar "RVal"]
                     ,Var (Short "RVal"))]
```

Figure 7.1: The new definition of `force` that makes use of the state to add call-by-need semantics.

```
fun force e =
  case e of
    val v => v
  | thunk ptr =>
      case !ptr of
        refVal v => v
      | refExp e =>
          let val _ = ptr := refVal (force (e 0))
          in case !ptr of
               refVal v => v
          end
```

Figure 7.2: The concrete syntax of the new `force` definition.

as delayed expressions are stored in the state instead of the environment. A different example is described in the following section that shows how call-by-need is tested.

7.3 Testing call-by-need

The testing of the call-by-need mechanics mainly consists of running the tests used in the creation of the `compile` function, by using `compareEval` described in Section 6.3. However, with the optimisation, the state is used for creating thunks. This means that evaluating certain compiled expressions causes the state to grow to a larger size than when evaluating the same expressions without compiling them. Thus, the state is omitted in the comparison of this testing phase, testing only the result of the evaluations.

Another factor that needs testing is how the thunks are changed in the state. As stated in Section 7.1, when thunks are created, they are created as delayed evaluations of expressions, represented as `RefExp`. When they are forced, they are changed to become values, represented as `RefVal`. In order to test that evaluated thunks indeed change to `RefVal` values, expressions that create and use thunks are evaluated with the `efc` function described in Section 6.3. Similarly, thunks that are not forced are checked to see if they remain as `RefExp` values after evaluation. As evaluation yields both the result and the state, the thunk references can be investigated to see if they are `RefExp` or `RefVal` constructors.

Call-by-need is also tested by using thunks that require a noticeable amount of time to evaluate. One method is to create a list of a replicated expression:

```
cakeReplicate =
  LetRec [("repeat", "elem",
            Fun "n" $
            If (App Equality [Var (Short "n"), Literal (IntLit 0)])
             (Con (Just (Short "nil")) [])
             (Con (Just (Short "::"))
              [Var (Short "elem"),
               App OpApp [App OpApp [Var (Short "repeat"),
                                     Var (Short "elem")],
                          decr]])
          )] (Var (Short "repeat"))
```

Here, `decr` is the same function as the one used for the definition of `take`. In concrete syntax, `replicate` can be defined as such:

```
fun repeat (elem,n) =
  if n=0 then nil
  else elem::(repeat (elem,n-1))
```

This function `cakeReplicate` is used to create a list with an expression that counts down to 0 and returns the string literal ”OK”:

```
stopAtZero =
  LetRec [("stopAtZero", "n",
            If (App Equality [Var (Short "n"), Literal (IntLit 0)])
             (Literal (StrLit "OK"))
             (App OpApp [Var (Short "stopAtZero"), decr])
           )] (Var (Short "stopAtZero"))
```

In concrete syntax, `stopAtZero` can be defined as such:

```
fun stopAtZero n =
  if n=0 then "OK"
  else stopAtZero (n-1)
```

Replication of `stopAtZero` is tested with two versions of the compiler: (1) with the call-by-need optimisation and (2) without call-by-need. For a list of `stopAtZero` that counts from 100, (1) prints the whole list much faster than (2), which requires a noteworthy amount of evaluation time for each list element. This result shows that call-by-need in (1) indeed works as intended: each thunk's suspended expression is evaluated once and replaced with a value, which is used in future calls, reducing the total evaluation time.

With the results of this testing phase being positive, the implementation of the call-by-need semantics is deemed to be finished. This also marks the completion of the compiler as a whole, meaning that all the coding has now been finished.

8

Future work

This chapter will discuss potential work that could be performed in addition to what has been done for this project. Suggestions will not only concern what the compiler takes as input and produces, but also tasks such as testing.

8.1 Compiling declarations

As stated in Section 2.2, expressions are the topmost construct of CakeML's grammar considered for this project. CakeML has a level of grammar above expressions: declarations. Declarations handle features such as multiple bindings at once and type definitions, and increase the expressiveness of the language. In order to support all of CakeML's semantics, defining declarations in the Haskell port of the semantics must be done. Lazy semantics and compilation of declarations would then need to be defined and implemented.

8.2 Property-based testing

For this project, the semantics and compiler are tested by using unit tests. As unit tests are dependent on the manual work of inventing and writing test cases, they cannot be fully trusted to be able to catch all types of errors. In the case of compiler construction, the expressiveness of the source and target languages can be tremendous, making it very difficult to think of all types of expressions to test. A different approach to testing is property-based testing. For Haskell, QuickCheck is a well known tool for formulating and running property-based tests [21]. It was suggested at the planning phase of this project that QuickCheck would be used to run property-based tests for the produced compiler. However, doing property-based testing on compilers was deduced to be a task that may warrant a project of its own. Thus, due to time constraints, property-based testing was not included for this project.

8.3 Update semantics

CakeML is a language that is being developed at the time of writing. As such, the version of semantics used to create the compiler for this project was already outdated shortly after the project start. If the compiler is to be used for the newest version of CakeML, the semantics would need updating. For constant support, the semantics would need to be continuously updated.

8.4 Applying the compiler to actual CakeML code

This project only handled the intermediate representation of CakeML's abstract syntax. An interesting application of the compiler would be to create a parser to produce code that can be executed through CakeML's own compiler. The parser would need to be able to take CakeML's concrete syntax, convert it to the Haskell port of the abstract syntax. Once the abstrax syntax has been generated, it can be run through the compiler to produce altered abstract syntax, which can then be converted back to the concrete syntax.

9

Conclusion

This thesis presented a Haskell definition of CakeML's strict semantics, a Haskell definition of lazy semantics for CakeML's pure parts, and a Haskell implementation of a semantics-preserving compiler that compiles lazy CakeML to strict CakeML.

The first step of this project, translating CakeML's semantics from Lem to Haskell, was performed successfully. Three items were transcribed to Haskell:

- The abstract syntax of CakeML.
- The helper functions for CakeML's semantics.
- The top level interpreter function `evaluate`.

The abstract syntax represents the grammar of the language, including literals, expressions, and operations. The highest level of the grammar that was translated was expressions, as it is the most important part of the language that allows operations and literals to be represented. The semantic helper functions of a language are the operations that are needed to produce a value. Some of these operations include arithmetics and pattern matching. Also a part of the semantic primitives are the representation of values, environment, and state. In order to go from an expression to a value, the `evaluate` function links the abstract syntax and the semantic primitives. Taking a state, a environment, and a list of expressions, `evaluate` applies the operations found in the semantic primitives to produce a state and a value.

The similarity in syntactical structure between the Lem and Haskell made the task of translating CakeML's semantics allowed the translation to be finished begin fairly early, which was followed by the second phase of the project: defining the lazy semantics. The task of defining the lazy semantics mainly consisted of creating a new definition of `evaluate`, called `evaluateLazy`, that targets the pure parts of CakeML. The lazy semantics used the idea of thunks to suspend evaluation of expressions until called. Thunks were defined under the value datatype in the semantic primitives and contained an expression

together with its environment. The function `evaluateLazy` was designed to evaluate an expression as little as possible before returning either a thunk with the partially evaluated expression and its environment, or another type of value. The function `force` was designed to *force* thunks to become usable values by repeatedly evaluating the delayed expressions until it produced a usable (non-thunk) value.

When the definition of the lazy semantics was finished, the third phase of this project began: constructing the compiler. This was formalised with a `compile` function that takes an expression and alters it with respect to the lazy semantics defined for this project. As such, thunks were defined by using the `Con` expression, which represents constructors. Together with the `Thunk` constructor, a `Val` constructor was also created to represent all non-thunk values. Values were simply put inside the `Val` constructor, while thunks required more processing. To simulate the encapsulation that thunks had in the lazy semantics, an expression was put inside a `Fun` expression to be evaluated to a closure. This would allow the expression to stay together with its environment in the `Thunk` constructor. In order for the compiled expressions to evaluate to usable values, a `force` function was defined. The idea behind the compiler's `force` was the same as the one defined for the lazy semantics, except that the compiler's `force` was defined by using CakeML expressions. By doing this, `force` could be injected into the compiled code to be evaluated together with the thunks and values, allowing non-thunk values to be produced at the end of the evaluation chain.

The implementation of the compiler allowed evaluation of expressions to be delayed until needed. However, delaying the evaluation only results in *call-by-name* semantics, meaning that if the thunks are used on multiple occasions, they will be evaluated each time. True laziness is achieved through *call-by-need* semantics where thunks retain the value yielded by the evaluation of their expressions. Call-by-need was implemented as an optimization by altering how thunks were produced and how `force` handled this new definition of thunks. Thunks were changed to contain a reference to a location in the state, where a suspended expression would be stored. Thus, `force` was change to extract the value that the reference pointed to, which would be represented by one of two new constructors `RefExp` and `RefVal`. Suspended expressions were represented by `RefExp`, while the values that the expressions were forced to were represented by `RefVal`. The change from `RefExp` to `RefVal` was handled by using an assign operation whenever a `RefExp` was forced to replace the value that the thunk reference was pointing at.

The ultimate goal of this project was to produce a compiler that takes lazy semantics and enforces it in strict semantics. This goal has been reached with the completion of the third phase. The call-by-need semantics optimized the compiler to achieve true laziness, where forced thunks would replace their suspended expressions with the value yielded from the evaluation.

Bibliography

[1] Lazy evaluation - HaskellWiki, https://wiki.haskell.org/Lazy_evaluation, (Accessed on 2017-05-05).

[2] Thunk - HaskellWiki, https://wiki.haskell.org/Thunk, (Accessed on 2017-01-19).

[3] J. Hughes, Why Functional Programming Matters, Computer Journal 32 (1989) 98–107.

[4] Z. J. Hu, J. Hughes, M. Wang, How functional programming mattered, National Science Review 2 (2015) 349–370.

[5] P. Z. Ingerman, Thunks: A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations, Commun. ACM 4 (1) (1961) 55–58.

[6] HOL Interactive Theorem Prover, https://hol-theorem-prover.org/, (Accessed on 2017-01-19).

[7] M. Fernández, Programming Languages and Operational Semantics, Springer London, 2014.

[8] R. Ennals, S. P. Jones, Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-strict Programs, SIGPLAN Not. 38 (9) (2003) 287–298.
URL http://doi.acm.org/10.1145/944746.944731

[9] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: principles, techniques, and tools, Second;Pearson new international; Edition, Pearson, Harlow, Essex, 2014.

[10] R. F. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, 1st Edition, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[11] R. Kulkarni, A. Chavan, A. Hardikar, Transpiler and it's Advantages.

[12] CakeML, https://cakeml.org/, (Accessed on 2017-04-13).

[13] R. Kumar, M. O. Myreen, M. Norrish, S. Owens, CakeML: A Verified Implementation of ML, in: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, ACM, New York, NY, USA, 2014, pp. 179–191.
URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2535838.2535841`

[14] Lem, `https://www.cl.cam.ac.uk/~pes20/lem/`, (Accessed on 2017-02-10).

[15] S. Owens, P. Böhm, F. Zappa Nardelli, P. Sewell, Lem: A lightweight tool for heavyweight semantics, Vol. 6898, 2011, pp. 363–369.

[16] seq - HaskellWiki, `https://wiki.haskell.org/Seq`, (Accessed on 2017-06-17).

[17] Performance/Strictness - HaskellWiki, `https://wiki.haskell.org/Performance/Strictness#Evaluating_expressions_strictly`, (Accessed on 2017-06-15).

[18] Lazy(T) Class (System), `https://msdn.microsoft.com/en-us/library/dd642331(v=vs.110).aspx`, (Accessed on 2017-06-15).

[19] O. Sturm, B. (e-book collection), E. C. (e-book collection), I. Books24x7, Professional functional programming in C#: classic programming techniques for modern projects, Wiley, Chichester, West Sussex, U.K, 2011.

[20] A. Ranta, M. Forsberg, Implementing programming languages: an introduction to compilers and interpreters, Vol. 16., College Publications, London, 2012.

[21] K. Claessen, J. Hughes, QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, in: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, ACM, New York, NY, USA, 2000, pp. 268–279.
URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/351240.351266`

A

Haskell definition of CakeML's abstract syntax

This appendix displays the Haskell definition of CakeML's abstract syntax.

```
-- | Literal constants
data Lit
  = IntLit Int
  | Char Char
  | StrLit String

-- | Identifiers
data Id a
  = Short a
  | Long ModN a

-- | Variable Names
type VarN = String

-- | Constructor Names
type ConN = String

-- | Type Names
type TypeN = String

-- | Type Variable Names
type TVarN = String
```

```
-- | Module Names
type ModN = String

-- | Type Constructors
data TCtor
  = TC_int
  | TC_char
  | TC_string
  | TC_fn

-- | Types
data T
  = TVar TVarN
  | TVar_DB Natural
  | TApp [T] TCtor

-- | Patterns
data Pat
  = PVar VarN
  | PLit Lit
  | PCon (Maybe (Id ConN)) [Pat]
  | PRef Pat
  | PTAnnot Pat T

-- | Expressions
data Exp
  = Raise Exp
  | Handle Exp [(Pat, Exp)]
  | Literal Lit
  | Con (Maybe (Id ConN)) [Exp]
  | Var (Id VarN)
  | Fun VarN Exp
  | App Op [Exp]
  | Log LOp Exp Exp
  | If Exp Exp Exp
  | Mat Exp [(Pat, Exp)]
  | Let (Maybe VarN) Exp Exp
  | LetRec [(VarN, VarN, Exp)] Exp
  | TAnnot Exp T
```

```
data Op
  -- Integer operations
  = OPN Opn
  | OPB Opb
  -- Function application
  | OpApp
  -- Reference operations
  | OpAssign
  | OpRef
  | OpDeref
  -- Char operations
  | Ord
  | Chr
  | ChOpb Opb
  -- String operations
  | Implode
  | StrSub
  | StrLen
  -- Vector operations
  | VFromList
  | VSub
  | VLength

-- | Built-in binary operations
data Opn
  = Plus
  | Minus
  | Times
  | Divide
  | Modulo

data Opb
  = Lt
  | Gt
  | LEq
  | GEq

data LOp
  = And
  | Or
```

B

Haskell definition of important datatypes of CakeML's semantics

This appendix displays the datatypes necessary to express CakeML's semantics. The full artifact that includes the helper functions can be found in the GitHub repository linked in Section 2.1.

```
data V
  = LitV Lit
  -- Constructor Application
  | ConV (Maybe (ConN, TId_or_Exn)) [V]
  -- Function Closures
  | Closure (Environment V) VarN Exp
  | RecClosure (Environment V) [(VarN, VarN, Exp)] VarN
  | Loc Natural
  | VectorV [V]
  | Thunk (Environment V) Exp

data TId_or_Exn
  = TypeId (Id TypeN)
  | TypeExn (Id ConN)

type AList k v = [(k, v)]
type AList_Mod_Env k v = (AList ModN (AList k v), AList k v)

data Environment v' = Env
  v :: AList VarN V,
  c :: AList_Mod_Env ConN (Natural, TId_or_Exn),
  m :: AList ModN (AList VarN V)
```

```
data State = St
  refs          :: Store V,
  defined_types :: S.Set TId_or_Exn,
  defined_mods  :: S.Set ModN


data Store_V a
  = RefV a
  | W8Array [Word8]
  | VArray [a]

type Store a
  = [Store_V a]

data Result a b
  = RVal a
  | RErr (Error_Result b)

data Error_Result a
  = RRaise a
  | RAbort Abort

data Abort
  = RType_Error
  | RTimeout_Error
```

C

Compiler

This appendix will display the Haskell code for the `compile` function, including helper functions used for the implementation. The code shown here is the post-optimization version of the compiler that has call-by-need semantics.

```
compile :: Exp -> Exp
compile (Raise e) = makeThunk $ Raise $ forceCompile e
compile (Handle e pes) = makeThunk $ Handle (forceCompile e) (compilePats pes)
compile (Con cn es) = makeVal $ Con cn $ map thunkCompile es
compile (Var n) = Var n
compile (Fun x e) = makeVal $ Fun x (compile e)
compile (Literal l) = makeVal $ Literal l
compile (App op es) = case (op, es) of
  (OpApp, [e1, e2])  ->
    makeThunk $ App op [forceCompile e1, thunkCompile e2]
  _ ->
    makeVal $ App op $ map forceCompile es
compile (Log lop e1 e2) =
  makeVal $ Log lop (forceCompile e1) (forceCompile e2)
compile (If e1 e2 e3) =
  If (forceCompile e1) (thunkCompile e2) (thunkCompile e3)
compile (Mat e pes) =
  Mat (forceCompile e) (compilePats pes)
compile (Let xo e1 e2) =
  Let xo (thunkCompile e1) (thunkCompile e2)
```

```
compile (LetRec funs e) =
  LetRec (recVals funs) (repl (compile e) (fst3 funs))
  where
    fst3 []     = []
    fst3 ((f,_,_):fs) = (f:fst3 fs)
    repl e []     = e
    repl e (f:fs) = Let (Just f) (makeVal (Var (Short f))) (repl e fs)
    recVals []           = []
    recVals ((f,x,e):xs) =
      (f,x,
       Let (Just f)
       (makeVal (Var (Short f)))
       (compile e))
      :(recVals xs)
compile (TAnnot e t) = TAnnot (thunkCompile e) t

forceCompile = force . compile
thunkCompile = makeThunk . compile

makeThunk :: Exp -> Exp
makeThunk e = Con (Just (Short "Thunk"))
              [App OpRef [Con (Just (Short "RefExp")) [Fun "" e]]]
makeVal :: Exp -> Exp
makeVal e = Con (Just (Short "Val")) [e]

makeRefVal :: Exp -> Exp
makeRefVal e = Con (Just (Short "RefVal")) [e]

mapPat :: (Exp -> Exp) -> (Pat, Exp) -> (Pat, Exp)
mapPat f (p,e) = (p, f e)

compilePat :: (Pat, Exp) -> (Pat, Exp)
compilePat  = mapPat compile
compilePats = map compilePat
```

```
force :: Exp -> Exp
force e =
  App OpApp [LetRec [("force", "exp"
                      , Mat (Var (Short "exp"))
                        [(thunkPat [PVar "TPtr"]
                         ,refMat (App OpDeref [Var (Short "TPtr")]))
                        ,(valPat [PVar "Val"]
                         , Var (Short "Val"))]
                      )] (Var (Short "force"))
             , e]

refMat :: Exp -> Exp
refMat e =
  Mat (e)
  [(refValPat [PVar "RefV"]
   ,Var (Short "RefV"))
  ,(refExpPat [PVar "RefE"]
   ,Let Nothing (App OpAssign [Var (Short "TPtr"), makeRefVal sol])
     (getVal (App OpDeref [Var (Short "TPtr")]))
   )
  ]
  where sol = App OpApp [Var (Short "force"),
                         App OpApp [Var (Short "RefE")
                                   ,Literal (IntLit 0)]]
        getVal e' = Mat e'
                    [(refValPat [PVar "RVal"]
                     ,Var (Short "RVal"))]

thunkPat ps = PCon (Just (Short "Thunk")) ps
valPat   ps = PCon (Just (Short "Val"))   ps

refValPat ps = PCon (Just (Short "RefVal")) ps
refExpPat ps = PCon (Just (Short "RefExp")) ps
```