CHALMERS

LLVM back-end for the Timber compiler

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and Logic

MATTIAS LUNDELL

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2010

LLVM back-end for the Timber compiler

MATTIAS LUNDELL

Examiner: BJÖRN VON SYDOW

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2010

Abstract

This master's thesis describes the design and implementation of a new back-end for the Timber compiler, based on *Low Level Virtual Machine* (LLVM) compiler infrastructure. The thesis describes the implementation and gives detailed information about how a Timber program is translated into a LLVM program and how it is represented. Furthermore the thesis compares the existing C based back-end with the new LLVM based back-end in terms of implementation complexity and the performance of programs that were compiled with the respective back-ends. When compared to each other, the LLVM based back-end was found to have a performance that was equal or better than the C based back-end, with the cost of increased complexity of the compiler in the form of an abstract syntax to represent LLVM assembly code and an interface to the abstract syntax. The overall conclusion is that LLVM offers a suitable back-end for the Timber programming language compiler.

Keywords: Timber, LLVM, Low Level Virtual Machine, Compiler back-end, Code generation

Sammanfattning

Denna uppsats beskriver design och implementation av en ny bakända till Timber kompilatorn. Den nya bakändan baseras på *Low Level Virtual Machine* (LLVM) som är en infrastruktur för kompilatorer. Uppsatsen beskriver implementationen och ger detaljerad information om hur Timberprogram översätts till LLVM program och hur dessa är representerade. Vidare görs en jämförelse mellan den befintliga C baserade bakändan, och den nya LLVM baserade bakändan i termer av komplexiteten av implementationen och prestandan hos program kompilerade med respektive bakända. Jämförelsen visade att den LLVM baserade bakändan var kapabel att generera kod med samma eller bättre prestanda än den kod som genererats med den C baserade bakändan. Detta till en kostnad av ökad komplexitet i kompilatorn i form av en abstrakt syntax för att representera LLVM assemblerkod, samt ett interface till den abstrakta syntaxen. Den slutgiltiga bedömmningen är att LLVM är lämplig som bakända till programmeringsspråket Timbers kompilator.

Nyckelord: Timber, LLVM, Low Level Virtual Machine, Kompilator bakända, Kodgenerering

Acknowledgements

I would like to thank Björn von Sydow for his support, for sharing his knowledge and for the many interesting discussions. Discussions that guided me in the intriguing world of compilers and kept me focused during the year I worked on the project.

Contents

Chapter 1

Introduction

Writing a compiler for a high-level programming language that produces efficient code for a specific architecture forces language designers to put a lot of work on instruction selection, register allocation and low-level optimizations performed on the generated code. If programs are going to be executed on an other architecture, the back-end must be rewritten to support the new architecture and the new set of instructions. To gain portability and to get away from implementing and maintaining a back-end for each architecture another approach must be taken.

One solution is to transform the code into another high-level language such as C, which is the case in the original Timber compiler. The reason for choosing C is that it operates on quite low level and there are compilers available on most platforms. On the negative side is that there are small possibilities to control the assembly code that is generated by the C compiler.

Another solution is to target a high-level assembly language. *Low Level Virtual Machine* (LLVM) is such a language, along with a compiler infrastructure that makes it possible to target multiple platforms by only generating code in the LLVM assembly format. In addition to being able to generate executable code for many platforms, LLVM also provides both platform independent and platform dependent low-level optimizations, which gives language designers room to only focus on the language design and front-end optimizations, while the back-end optimizations are for free.

This thesis describes the implementation of a new back-end for the Timber programming language compiler based on the LLVM compiler infrastructure. It will also cover a comparison between the existing C back-end and the implemented LLVM back-end in terms of the performance and the size of programs compiled with the two different back-ends.

1.1 Aim

The goal with the thesis work was to implement a new back-end for the existing Timber compiler based on LLVM. Along with a description of the implementation, this thesis also tries to answer the following questions:

- Is LLVM a suitable target for the implementation of a back-end for a functional language and how complex does the back-end become in comparison to the existing C back-end?
- How does the quality of the code generated by the LLVM based back-end compare to the code generated by the existing C back-end measured in execution time and size of generated files?

1.2 Limitations

- The compiler has no support for handling LLVM related optimization flags.
- The comparison between the LLVM back-end and the C back-end is rudimentary.

1.3 Thesis outline

The thesis is divided into six chapters. Chapter 1 is this introductory overview. Chapter 2 gives an introduction to the programming language Timber. Chapter 3 gives an introduction to LLVM virtual instruction set and the LLVM infrastructure. Chapter 4 covers the implementation of the LLVM back-end. Chapter 5 contains a comparison of the characteristics of programs compiled with the new LLVM back-end and with the original C back-end. Finally, chapter 6 concludes the report and discusses future work.

Chapter 2

Timber

Timber is a general programming language specifically aimed at the construction of complex event-driven systems. These event-driven systems are ranging from low-level device interfaces, over time-constrained embedded systems, to very high-level symbolic manipulation and modeling applications. The language is developed and maintained by groups and individuals at Luleå University of Technology, Chalmers University of Technology, University of Kansas and Portland State University.

2.1 Language summary

The language Timber can be seen as a successor of O'Haskell [8] which is a reactive, object-oriented, concurrent, functional language implemented as an extension to Haskell. This relation is visible in the syntax of Timber, which is very similar to the syntax of Haskell.

One big difference between O'Haskell and Timber is that O'Haskell had lazy semantics, but in Timber the lazy semantics is switched to strict semantics. The main reason for shifting from lazy to strict semantics was that lazy semantics adds extra difficulties to time and space analysis of programs, an analysis which is of importance when constructing real-time and embedded software [2, p. 6].

Timber is a multi-paradigm programming language that tries to combine the best features from three different programming paradigms.

- It is a pure functional programming language similar to Haskell. It supports recursive definitions, higher-order functions, algebraic datatypes, pattern matching and Hindley/Milner-style polymorphism. Timber also adds two features, subtyping and a monadic implementation of reactive objects.

- It is an imperative object-oriented programming language with stateful objects supporting subtyping. It also has common features from imperative languages such as loop constructs and variable assignment.
- It is also a strongly typed concurrent programming language with message based interaction supporting both synchronous and asynchronous communication. In Timber this is represented by a set of interconnected reactive objects, each encapsulating a piece of the current global state. The objects all evolve in parallel but execution of methods belonging to a certain object are mutually exclusive. This makes concurrency implicit in Timber and no notion of threads or other concurrency constructs are needed.

2.2 Reactive objects

The main building blocks of a Timber program are reactive objects. A reactive object consists of a set of methods and a set of state variables. The methods are invoked by external events or clocks. During idle time, an object remains passive, maintaining its state and waiting for next event.

The interface to an object is typically described by a record with the methods that the environment can invoke, as in this example:

```
struct Counter where
    incr  :: Action
    value :: Request Int
```

This interface defines two methods. The first method `incr`, has type `Action` which indicates that the method is asynchronous, the other method `value`, is of type `Request Int` which means that the method is a synchronous method returning a value of type `Int`.

The class itself consists of a set of state variables and the implementation of the methods defined in the interface.

```
counter = class
    n := 0
    incr  = action n := n+1
    value = request result n
    result Counter{..}
```

Instances of the class `counter` maintain a state that consists of an integer `n`. The state can be modified by the asynchronous method `incr`, and the state can be retrieved with the synchronous method `value`.

2.3 Time in Timber

One of the main features of Timber is that time is incorporated in the language and deadlines can be expressed explicitly. Time is described in terms of absolute time, which means that time progresses independently from program execution. Programs can not themselves express or access absolute time, but they can measure durations of time intervals. These durations are captured in a primitive type `Time`, which measures the length of time intervals that may occur in a program. `Time` is an instance of the type class `Num` which makes it possible to add and subtract values of type `Time` (multiplication is undefined).

2.3.1 Timing window

In Timber, each method execution is associated with a timing window delimited by a baseline and a deadline, both in terms of absolute time. The baseline acts as the starting point of a reaction and may also be used as a reference point in time, to which time expressions may relate. The deadline is the time where the system must have reacted to the event in order for the system to behave correctly. The timing window is by default inherited from the method that initiated the execution, but for methods of type `Action` it is possible to manually specify the timing window. This enables the programmer to specify timing constraints of a Timber program in a platform-independent manner. The manipulation of the timing window is performed with two constructors, `after` and `before`.

```
after (sec 10) act
```

This expression sets the baseline for the expression `act` to be the current baseline plus 10 seconds.

```
before (sec 10) act
```

This expressions sets the deadline for expression `act` to be its effective baseline plus 10 seconds. By combining `after` with a recursive call it is possible to express periodic execution

```
fun = action
        complexCalculation
        after (sec 10) fun
```

2.4 Programming in Timber

To give an illustration of how a Timber program may look, an implementation of a prime number sieve based on the *Sieve of Eratosthenes*[1] is given.

```
module Sieve where
```

A Timber program is built from a set of modules. One of these modules is the root module which contains the definition of the root function. The root function is the starting point for the execution of a Timber program. The other modules contains definitions of functions used by the root module and by other auxiliary modules.

```
import POSIX
```

After the header of the module all import declarations are listed. In this case the program is intended to run under the POSIX environment[2], therefore the module `POSIX` is imported. The `POSIX` module contains an interface to methods from the environment that are available, such as access to standard output which can be used to print strings to the terminal.

```
struct Cell where
    feed :: Int -> Action
```

In Timber, interfaces are implemented as structs which defines methods that are reachable from the outside of an object. In this example an object of type `Cell` has one asynchronous method visible, `feed` that takes an `Int` as input and returns an `Action`.

```
cell print n = class
    next := Nothing

    feed k = action
      if (mod k n) /= 0 then
        case next of
          Nothing -> do
            c = new cell print k
            next := Just c
            print (show k ++ "\n")
          Just c -> do
            c.feed k

    result Cell{..}
```

Timber class definitions start with the keyword `class`. Classes consist of local state variables and their initializations, declarations of class methods

[1] The program is a rewrite of an example written in O'Haskell given in [8, p. 65]

[2] **P**ortable **O**perating **S**ystem **I**nterface for Uni**x** [4]

that access and modify the state, local auxiliary definitions and a result. Typically the result is one or more interfaces that the class implements. In this example there is a single state variable `next`, that holds the next cell in the computation chain. There is also one class method `feed` and a result that returns a value of type `Cell`.

```
sieve print n = do
   c = new cell print 2
   forall i <- [3..n] do
       c.feed i
```

The function `sieve` takes a printing procedure and an integer `n`, a new cell is created using the keyword `new` and for each number `i` from 3 to `n`, `i` is tested for primality using the cell chain.

```
root :: World -> Cmd () ()
root w = do
   env = new posix w
   sieve env.stdout.write 1000
```

Finally the root function of the module. The root function takes a single argument of type `World`, an abstract type representing the world external to the program. In this example the program is executed under the POSIX environment and in that environment the function `stdout.write` is defined which is used to print strings to the terminal.

Chapter 3

LLVM

The Low Level Virtual Machine (LLVM) is a open-source compiler infrastructure, a virtual instruction set and a compilation strategy. The compilation strategy of LLVM is to support effective program optimization during the entire lifetime of a program. LLVM supports optimizations at compile-time, link-time, run-time and idle-time[1][7]. The development of LLVM started in 2000 as a part of Chris Lattner's Master thesis[5]. The primary components of the infrastructure is a GCC-based front-end for C and C++ called llvm-gcc, a link-time optimization framework, Just-In-Time compilers and static back-ends for several architectures. In the current version 2.7 the following architectures are fully supported:

ARM	Mips	Mipsel
MSIL	PowerPC 32	PowerPC 64
Sparc	SystemZ	Thumb
x86	x86-64	XCore

Table 3.1: Architectures supported by LLVM 2.7.

Other architectures are also supported but on an experimental level e.g. the Cell CPU[6].

3.1 Language

LLVM code representation is designed to be used in three different forms: as an compiler IR, as bitcode representation written to a file and as an assembly

[1]Optimizations between runs, using profile information collected during runs.

language. The three forms of representation is all equivalent. This section gives an introduction of the LLVM assembly language and examples on how the assembly language is used.

3.1.1 SSA

LLVM uses Static Single Assignment (SSA) form as its primary code representation. To be in valid SSA form, each variable can only be defined by a single definition [3, p. 228]. If the same variable is defined in two different control-flow paths the rule is broken. To solve this problem the notion of ϕ-functions is introduced. The ϕ-function returns the value of the argument that corresponds to the control-flow path that was taken.

```
if (even) then x := 0 else x := 1
return x
```

In this example the variable `x` is defined in two different control-flow paths. To transform the example into SSA form the ϕ-function must be used.

```
if (even) then x1 := 0 else x2 := 1
x := φ(x1, x2)
return x
```

The resulting value returned by ϕ`(x1, x2)` depends on which of the paths the control flow takes, if it passes through the true branch of the `if` statement the resulting value is `x1` and if passes through the false branch the resulting value is `x2`. To support SSA form LLVM provides an infinite number of virtual registers and an instruction named `phi`.

```
<result> = phi <type> [ <val0>, <label0> ], ...
```

The semantics of `phi` is that `result` will take the value `vali` if the current basic block is preceded by the basic block with label `labeli`.

3.1.2 Instruction Set

The LLVM instruction set is designed to capture the key operations of ordinary processors without any processor specific constraints such as the number and type of physical registers[7]. Instead of physical registers, the instructions operate on virtual registers. The LLVM instruction set consists of eight different groups of instructions.

Terminator Instructions

In LLVM, every basic block must end with a terminating instruction. Terminating instructions indicate which basic block that will be executed after

the current block. The terminator instructions used in the LLVM back-end is: `ret`, `br` and `unreachable`.

Binary Operations

The binary operations take two operands and perform an operation on them, the result is a single value. Both operands are required to have the same type, the result from the computation has the same type. The operators operate on values of integer type, floating point type and vectors. Examples of binary instructions used in the LLVM back-end are: `add`, `fadd` and `mul`.

Bitwise Binary Operations

Bitwise operations perform bit manipulation. Both arguments to the operation have to be of the same integer type. The operations can also work with vectors of integer type. The result of the computation has the same type as the operands. Examples of bitwise binary instruction used in the LLVM back-end are: `and`, `or` and `shl`.

Vector Operations

In addition to the arithmetic binary operations and bitwise binary operations on vectors, LLVM offers instructions to extract respectively insert a scalar element into a vector at a specified index. The vector operations used in the LLVM back-end are: `extractelement` and `insertelement`.

Memory Access and Addressing Operations

The memory in LLVM is not in SSA form, but all accesses are made with instructions which are in SSA form. To transfer data between memory and the virtual registers, LLVM provides instructions to read, write and allocate memory. To address sub-elements inside aggregated data structures such as arrays and structures the instruction `getelementptr` is used. `getelementptr` does not access the memory, it only calculates the address to the specific element. To illustrate how to address the second structure field in a structure with three 32 bit integers.

```
%regptr = getelementptr {i32, i32, i32}* %sptr, i32 0, i32 1
```

Where `%regptr` is the identifier of the virtual register where the address to the indexed element are stored, `{i32, i32, i32}*` is the type of the structure, `%sptr` is the identifier of the virtual register where a pointer to the structure is stored and the following list of comma-separated 32 bit integers

are indexes used for pointing out the structure element of interest. The first index `i32 0`, tells the compiler to start address calculation from the address where the structure object is located. The second index `i32 1`, points to the second field in the structure object. The type of the virtual register `%reg` is a pointer to the type of indexed element, in this case `i32*`, a pointer to a 32 bit integer. To get the actual integer the instruction `load` is used.

```
%reg = load i32* %regptr
```

This loads the actual value into `%reg`. To store a value into `%reg`, the instruction `store` is used.

```
store i32 42, i32* %regptr
```

The `getelementptr` instruction is often seen as confusing[10].

Conversion Operations

There is only one way to convert a value of one type into a value of an other type and this is through a set of type conversation operations[7]. If for example a signed 32 bit integer is needed to be converted into float, the instruction `sitofp` is used.

```
%float = sitofp i32 %int to float
```

`sitofp` converts the integer stored in the virtual register `%int` into a float and stores the result in the virtual register `%float`, the resulting register has the type `float`. The operators that converts between integer and float types can also operate on entire vectors of values.

Other Operations

In this category are the "miscellaneous" instructions. The ones that are used in the Timber back-end is `icmp`, `fcmp` and `call`. The first two instructions are used to compare two values of integer type respectively floating point values. The result of a comparison of two values is a one bit integer. If the two operands are vectors the result is a vector of one bit integers. It is also possible to compare two pointers against each other. The `call` instruction represents a function call. All functions that are called must have been defined with their type.

3.1.3 Type System

LLVM is a strictly typed representation. Every virtual register and memory location has a specified type. The typed information gives the LLVM compiler the ability to perform high-level transformation on low-level code. The

type system is divided into two parts, the primitive types and the derived types. The types are also divided into five different classes: integer, floating point, first class, primitive and derived.

Primitive types

The primitive types are the fundamental building blocks of the LLVM system.

Type	**Syntax**	**Description**
Integer	`iN` where `N` is a literal integer	An integer type with bit width `N`.
Floating point	`float`	32 bit floating point number, LLVM supports other floating point formats as well but in the Timber back-end only `float` is used.
Label	`label`	The label type represents code labels.
Void	`void`	The void type does not represent any value and has no size.

Table 3.2: LLVM primitive types.

Derived types

Derived types allow a programmer to represent types such as arrays, vectors and pointers. The derived types are build up by primitive types and other derived types.

Type	Syntax	Description
Array	`[10 x i32]`	An array with ten 32 bit integers.
Function	`i32 (i32)*`	A function that takes a 32 bit integer as an argument and returns a 32 bit integer.
Pointer	`i32*`	A pointer to a 32 bit integer.
Structure	`{i32, i32 (i32)*}`	A structure with a 32 bit integer in the first field and a function in the second.
Vector	`<3 x float>`	A vector with three 32 bit floating point numbers.
Opaque	`opaque`	Unknown content, used for example as a placeholder in a structure.

Table 3.3: LLVM derived types.

3.2 LLVM tools

LLVM provides a number of command-line tools. Some of them are used in the LLVM part of the Timber compiler.

llvm-as is the LLVM assembler, translating files written in LLVM assembly language into binary files containing LLVM bitcode. Before translating, the assembler ensures that the code is in SSA form and that all basic blocks ends with a terminator instruction.

llvm-ld is the LLVM linker. The linker takes a set of bitcode files and links them together into a single LLVM bitcode file. The linker is also capable of doing link-time optimizations. The llvm-ld tool is used by the Timber compiler to link programs and generate the native executable.

opt is the modular LLVM optimizer and analyzer. The tool analyzes and performs optimizations on single files in LLVM assembly format or in LLVM bitcode format. The result from the optimizer is either stored in a file using the bitcode format or printed out on the standard output.

llc is the LLVM static compiler, it compiles LLVM bitcode into native assembly language for a specified architecture. The tool is indirectly used by llvm-ld when generating the native executable.

Chapter 4

Implementation

This chapter describes the design and the implementation of the LLVM back-end of the Timber compiler. The chapter also describes how the different parts of a Timber module are translated and represented.

4.1 Compiler pipeline

The original Timber compiler generates C source code using a pretty printer that turns a Kindle module into one C source file and one header file. These files are compiled into object code using GCC, which is linked to the run-time system and the standard library again using GCC.

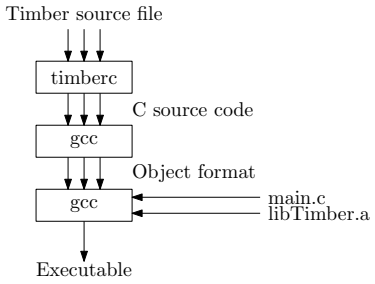

Figure 4.1: Compile chain with the C back-end.

The LLVM generating module is implemented as an additional Haskell module on the same level as the C generating module. The back-ends are separate from each other and the C back-end remains untouched. Both the LLVM back-end and the C back-end operate on the same input, Kindle modules.

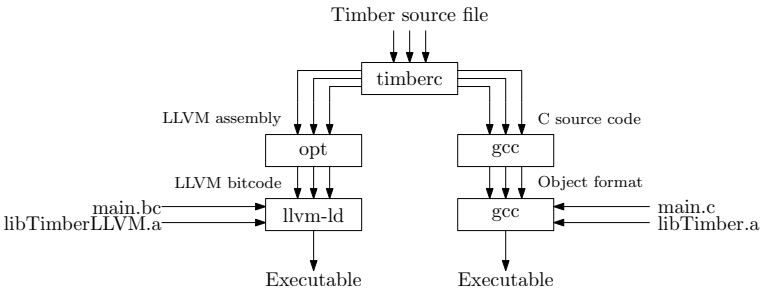

Figure 4.2: Compile chain with the LLVM back-end.

The LLVM code generator generates LLVM assembly code which is written to a file. Each module is translated into one file. Each LLVM assembly file is translated into LLVM bitcode representation and optimized using the LLVM optimizer *opt*. The resulting bitcode files are linked together with a main module named *main.bc*, the run-time system and the standard library using the LLVM linker *llvm-ld*. Both the run-time system and the standard library is located in an archive called *libTimberLLVM.a* that is created with LLVM archiver *llvm-ar*. The linker finally generates a native executable file using the LLVM system compiler *llc* and the system linker internally.

4.1.1 Run-time system

The run-time system of Timber is written in C. The only change that was made to the run-time system was to add an additional function `new`. The function provides the same functionality as the macro `NEW`, that is used in the C back-end for allocating memory on the heap managed by the run-time system. When used together with the LLVM back-end, the run-time system is compiled with Clang. Clang uses LLVM as its back-end and it is possible to compile into bitcode files which enables whole program optimizations with both the program, the standard library and the run-time system in the scope.

4.2 LLVM back-end

Before a program written in Timber is compiled into LLVM assembly code it passes through three different intermediate languages; “Syntax”, “Core” and “Kindle”. The first representation, Syntax is used during desugaring and renaming passes, the second language Core is the primary intermediate language and is used during kind analysis, type checking and term reduction. The third language Kindle is the main back-end intermediate language and

is used during tail call elimination and lambda lifting. Kindle is also the input to the LLVM code generator.

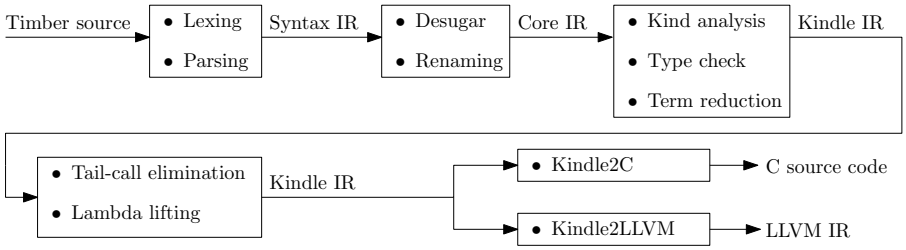

Figure 4.3: Internals of the Timber compiler.

4.2.1 Kindle

As said in section 4.3, Kindle in the main back-end intermediate language. It is a typed imperative language with support for dynamic memory allocation and garbage-collection. The language can be seen as an extended version of the common subset of C, Java and C++, without the pointer arithmetic in C and without the class hierarchies in Java and C++. In Kindle, a heap allocated structure object may contain function valued fields that are invoked by self-application. This gives Kindle a basic form of object-orientation capabilities. Kindle is intended to function as a high-level format that can be translated into standard imperative languages such as C and into an assembly language such as LLVM, which is done as a part of this thesis work.

4.2.2 Code generation environment

During code generation two environments are maintained. The first is on module level and contains information about the current module. The other contains information used in the current function.

Module level environment

This environment contains information that is common to the whole module. The content of the environment is

- The name of the current module.
- A mapping from names of global variables in imported modules to their virtual register.

- A mapping of names of top-level constants to their virtual register. Examples of constants are strings and arrays that are used by the garbage collector.
- A mapping from function names to their type. Both locally defined functions and functions from imported modules are present in the map.
- A collection of all external functions. This is used when generating function calls but it is also emitted to the final LLVM module as the LLVM assembler needs to have function declarations of all external functions that are used in the module.
- A collection of the functions that have been generated.
- A mapping between Kindle structures and named LLVM structure type definitions.
- A map containing all structures and information about their fields. This is used when looking up the type of a specific field and when generating the tags used by the garbage collector.
- The next fresh string identifier. LLVM does not permit string literals and therefore all string needs to be predefined and stored as a top-level constant with unique virtual registers.

Function level environment

This environment contains information that is used in the generation of a single function.

- The name of the function.
- The next fresh virtual register identifier.
- The next fresh label identifier.
- A list of labels that refer to the test part of a while-loop, this is used when generating code for the "continue" command.
- A list of labels that refer to the end part of a while-loop, this is used when generating code for the "break" command.
- A collection of the code that has been generated so far.
- A mapping between declared global and local variables and their virtual register.

4.2.3 Interface between Haskell and LLVM

To be able to interface between the Timber compiler front-end and the LLVM back-end, the front-end needs to be able to produce code that is readable by the LLVM machinery. The LLVM FAQ[12] suggest three different approaches to interface with LLVM:

1. Call into the LLVM libraries code using your language's FFI (foreign function interface).
2. Emit LLVM assembly from your compiler's native language.
3. Emit LLVM bitcode from your compiler's native language.

Due to the fact that the Timber compiler is implemented in Haskell and Haskell already has bindings to the LLVM API[1], the first approach was the first to be considered. But after a few tryouts with the bindings and a discussion with one of the maintainers (Lennart Augustsson), this approach was dropped. The main reason was that the bindings are operating on a very high level and tries to bring the type safety of Haskell into the API. This makes the API quite cumbersome to work with, taken into account that the only thing that is needed for the Timber back-end is the ability to generate code that is accepted by the LLVM compiler infrastructure.

The only reason for choosing approach number three instead of approach number two is that it would result in slightly faster compilation times, but the cost for that improvement is the tedious work of implementing the bitcode writer. Approach number three was therefore rejected in favor of approach number two, emitting LLVM assembly from Haskell. This approach was also taken by both *Essential Haskell Compiler*[9] and *Glasgow Haskell Compiler*[11].

4.2.4 LLVM abstract syntax

As said in section 4.2.3, LLVM assembly code is emitted by the LLVM back-end. LLVM assembly code is internally represented as abstract syntax implemented using algebraic data types (ADT) and generalized algebraic data types (GADT). The implementation of the abstract syntax is in the file "LLVM.hs". Here follows an description of the three most important parts, modules, values and instructions. A LLVM module is represented by

```
data LLVMModule = LLVMModule {
                    modName     :: String,
                    modTypDef   :: [LLVMStructDef],
                    modGlobal   :: [LLVMValue],
```

```
                    modFunDecl  :: [LLVMFunctionDecl],
                    modTopConst :: [LLVMTopLevelConstant],
                    modFuns     :: [LLVMFunction] }
```

where `modName` is the name of the module, `modTypeDef` is a list of LLVM type aliases to structure types, `modGlobal` which is a list of global variables (both local and from imported modules), `modFunDecl` which is a list of declarations of functions from imported modules, `modTopConst` which is a list of global constants such as strings, `modFuns` which is the functions defined in the current module.

LLVM values are represented with `LLVMRegister`s and `LLVMConstant`s represent the operands of an instruction. `LLVMRegister` is also used to store the result of an instruction. Virtual registers can be either local or global.

```
data LLVMValue = LLVMRegister LLVMType String LLVMRegisterTag
               | LLVMConstant LLVMType ConstValue
                 deriving (Eq,Show)

data LLVMRegisterTag = TagGlobal [LLVMLinkage] (Maybe LLVMGlobalInit)
                     | TagLocal
                       deriving (Eq, Show)
```

LLVM instructions are implemented as a GADT with one constructor for each instruction. As an example we have the instruction `Add`.

```
data LLVMInstruction where
    Add :: LLVMValue -> LLVMValue -> LLVMValue -> LLVMInstruction
    ...
```

4.3 Code generation

The translation of a Kindle module into a LLVM module is divided into four steps. First each Kindle type declaration is translated into a LLVM structure type and a description of the structure is inserted into the module environment described in 4.2.2. Secondly value binds are translated into global variables and inserted into the list of global variables in the current module. Thirdly all function bindings are translated into LLVM functions and inserted into the list of functions in the current module. Finally, a function that initiates the module is generated.

4.4 Representation of Data types

When generating LLVM assembly code to represent a Kindle program there are several different kinds of types that have to be handled.

- Primitive scalar types such as `Int` and `Float`.
- Primitive types such as `Array` and `List`.
- Data types that are introduced by the user using `data` declaration or `struct` declaration.

4.4.1 Kindle built-in types

Scalar Kindle types

Implementation of the Kindle primitive scalar types in LLVM is straightforward, each primitive scalar type is mapped onto corresponding primitive type in LLVM (except `OID` which is mapped to a pointer to a memory location).

Kindle type	LLVM type
`Int`	`i32`
`Float`	`float`
`Bool`	`i1`
`WORD`	`i32`
`OID`	`i32*`
`Char`	`i8`
`BITS8`	`i8`
`BITS16`	`i16`
`BITS32`	`i32`
`TUP0`	`i8`

Table 4.1: LLVM mapping of Kindle scalar types.

POLY

In the last step before the Kindle module is fed into the code generator the module passes through a function that replaces all type variables with a special type constructor `POLY`. In the LLVM back-end the type `POLY` is represented by a pointer to a memory location, holding a value of arbitrary type.

String

In Kindle, as well as in Timber, a string is a list of `Char`. To avoid that the generated code gets bloated with code representing the strings as lists of `Char` a helper function `getStr` is used. The function takes a C-style string

and returns a list of characters. To achieve this in LLVM all string literals are stored as constant arrays of 8 bit integers and when the string is needed a pointer to the first element of the constant array is passed as argument to `getStr` and the list representation of the string is returned.

```
@str0 = private constant [5 x i8] c"LLVM\00"
```

Listing 4.1: String representation in LLVM.

List

Kindle lists are implemented as single linked lists. The base type `%LIST` is used when lists are passed as arguments to a function or when a list is the return type of a function. Before a list is used inside a function it is compared against null to see if the list is empty or not. If the list isn't empty the list is cast to the `%CONS` type, indicating that the list is a cons-cell. The `%CONS` type consists of three fields, the first is a pointer to the garbage collector tag, the second field is the element or a pointer to the element and the third field is a pointer to the next element of the list (or null if there is no next element).

```
%LIST = type {i32*}
%CONS = type {i32*, i32*, %LIST*}
```

Listing 4.2: Kindle list type in LLVM.

Array

The array type is a structure with three fields, the first is the garbage collector tag, the second is an 32 bit integer that holds the length of the array and the third field is an array of unknown length that holds the actual data. If the data is of scalar type, such as `Int`, the actual values are stored in the array but if the data is heap allocated object a pointer to that object is stored in the array.

```
%ARRAY = type {i32*, i32, [0 x i32*]}
```

Listing 4.3: Kindle array type in LLVM.

Closure

Closures are implemented as four different structures based on the arity of the function. There is one for unary functions, one for binary, one for ternary and one for n-ary. Basically the structures consists of two fields, the first is a pointer to the tag used by the garbage collector and the second is a pointer to a function.

```
%CLOS1 = type {i32*, i32* (%CLOS1*, i32*)*}
%CLOS2 = type {i32*, i32* (%CLOS2*, i32*, i32*)*}
%CLOS3 = type {i32*, i32* (%CLOS3*, i32*, i32*, i32*)*}
%CLOS  = type {i32*, void ()*}
```

Listing 4.4: Kindle closure types in LLVM.

The function types all follow the same pattern, the first argument is the closure object itself and the rest of the arguments are arguments that are passed to the function stored in the closure. Closures with an arity of one, two or three are used as they are without typecast but closures with an arity greater than three must be typecasted before they are called. As an example we have a Timber function `f`

```
f :: (a -> a -> a -> a -> a) -> a -> a -> a -> a -> a
```

When compiled into LLVM the function gets the type

```
i32* @f(i32 %x, %CLOS* %g, i32* %a, i32* %b, i32* %c, i32* %d)
```

where the argument `%g` is the closure and `%a`, `%b`, `%c` and `%d` arguments to the closure. Before the closure is called it has to be typecasted from `void ()*` into a function type that takes the right number of arguments and returns the right type. The typecast is performed with the LLVM instruction `bitcast`

```
%r0 = load %CLOS** %g
%r1 = getelementptr %CLOS* %r0 , i32 0, i32 1
%r2 = load void ()** %r1
%r3 = bitcast void ()* %r2 to i32* (%CLOS*, i32*, i32*, i32*, i32*)*
```

This instruction sequence converts a function pointer of type `void ()*` into the type of a function that takes four arguments of type `POLY` and returns a value of type `POLY`. The result is a pointer to a function of correct type that is accessible from the virtual register `%r3`. The actual function call is then made by using the function pointer in `%r3` (the instruction is split across two lines).

```
%r4 = call i32* (%CLOS*, i32*, i32*, i32*, i32*)*
      %r3(%CLOS* %r0, i32* %a, i32* %b, i32* %c, i32* %d)
```

Ref

Ref is the type of a reference to an object, i.e. an instantiation of a class. The reference type divided into two parts, one is the locking mechanism and the other is the object. Under the POSIX environment the threading is performed using *pthreads* and the locking mechanism used in references is a union type `pthread_mutex_t` that is defined "pthreadtypes.h". The lock is located in the second field of the reference structure. The third field consists of a pointer to the object.

```
%REF = type {i32*, %PTHREAD_MUTEX_T, i32*}
```

Listing 4.5: Kindle reference type in LLVM.

When a class is instantiated the memory needed for that object is claimed using the primitive function `new`. The amount of memory needed is the combined size of both the Ref structure and the structure of the object that is referenced.

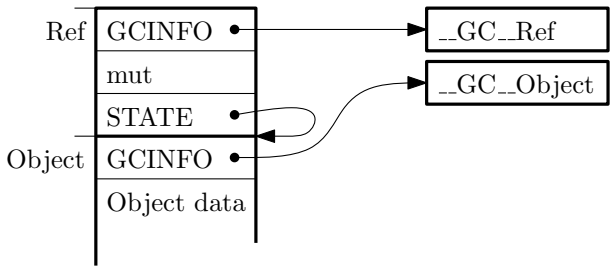

Figure 4.4: Memory layout of a Ref object.

The reason for the combined size is that the object that the reference refers to is located just after the reference itself. When the memory is allocated the mutex is initiated using the primitive function `INITREF`.

```
%reg1 = alloca %Ref*
%reg2 = bitcast %Ref** %reg1 to i32**
call void @new(i32** %reg2, i32 10)
%reg3 = load %Ref** %reg1
call void @INITREF(%Ref* %reg3)
```

Along with `INITREF`, references also have three other primitive functions that are used when references are involved; `ASYNC` that is used when invoking functions that have Timber type `Action`, `LOCK` that locks the object from being modified by other threads and the function `UNLOCK` that unlocks an object.

Other

The other built-in types, Timer, Msg, Tuple and Either are all straightforwardly translated into LLVM using structure types.

4.4.2 User defined type declarations

In Kindle a type declaration introduces a structure type that defines the layout of a heap-allocated object. The structure type may contain both value fields and function fields. The structure types are straightforwardly

translated into LLVM structure types. As an example of the translation we define a data type of lists parametrized over a type `a` in Timber.

```
data List a = Nil | Cons a (List a)
```

Listing 4.6: Timber representation of the user defined List data type.

This data type is translated into a set of Kindle structure types. Before the Kindle code is passed to the LLVM code generator the type parameter `a` is translated into a type constructor called `POLY`.

```
struct List {
    POLY GCINFO;
}
struct _Nil {
    POLY GCINFO;
}
struct _Cons {
    POLY GCINFO;
    POLY a;
    List b;
}
```

Listing 4.7: Kindle representation of the user defined List data type.

The first structure type is consists of a single field of type `POLY` and is used as a pointer to the head of the list, the second structure type `_Nil`, represents the empty list. Finally the third structure type `_Cons` represents a cons cell and contains three fields. The first field named GCINFO, contains a tag used by the garbage collector, the second field contains the actual element of the cell and the third field contains a pointer to the tail of the list. These three Kindle structure types are represented as type aliases to structure types in LLVM.

```
%List_Mod  = type {i32*}
%_Cons_Mod = type {i32*, i32*, %List_Mod*}
%_Nil_Mod  = type {i32*}
```

Listing 4.8: LLVM representation of the user defined List data type.

4.4.3 Garbage collection tag

The first field in all heap allocated objects contains information used by the garbage collector. The field is called GCINFO, and consists of a pointer to an array of 32 bit words describing the heap layout of the object. The array consists of $3+n$ words where n is the number of fields in the structure except the GCINFO field. The layout of the array is on the form

- The first element contains the size of the object in words.
- The second element contains a tag that classifies the object into a small number of groups.
- The next n elements contains either the offset in words for each field of the object that is a pointer or 0 if the field consists of a scalar.
- The last element is used when collecting arrays and indicates whether the arrays stores values of scalar types or if it stores pointers to heap allocated objects.

Continuing on the example of the List data type described in Listing 4.6. If the list contains elements of the scalar type `Int` the GCINFO tag is:

```
{3, 0, 2, 0, 0}
```

This array tells the garbage collector that the object has a size of three 32 bit words, it is of the standard structure type and that the element in the field with an offset of two 32 bit words is a pointer, in this case the pointer points to the next cons cell in the list. If instead the list contains elements of type `Tuple`, the GCINFO is:

```
{3, 0, 1, 2, 0}
```

Now the third and fourth element indicate that there are pointers in the fields with an offset of one and two, in this case the first pointer points at the tuple that is the element in the current cons cell and the second pointer points at the next cons cell in the list.

In the list case there is one type parameter and therefore two different GCINFO tags are needed, one is used when the element is of scalar type and one if the element is a pointer to a heap allocated object. In the general case where the data type has m type variables, 2^m different GCINFO tags are needed, which each consists of an array with $n + 3$ elements.

The Timber compiler generates one global array of $2^m(n + 3)$ words for each structure type. This array contains all 2^m different GCINFO tags each with a length of $n + 3$. When a new object is created their GCINFO field is set to point in a suitable sub array inside holding the correct tag depending on what type parameters that was used. In LLVM this array is implemented as a constant global array holding 32 bit integers.

```
@__GC__List_Mod  = global [3 x i32] [i32 1, i32 0, i32 0]
@__GC___Cons_Mod = global [10 x i32] [i32 3, i32 0, i32 1, i32 2, i32 0,
                                      i32 3, i32 0, i32 2, i32 0, i32 0]
```

Listing 4.9: Garbage collector tag for an object of type List

If we want to allocate a list object holding a single `Int` 5, the GCINFO field must be set to point at the second sub array of the global array `_GC___Cons_Mod`. In LLVM this is achieved by casting the address of the global array into an 32 bit integer and perform a sequence of arithmetic operations on the address and then typecast it back to a pointer again.

```
%reg2 = bitcast %List_Mod** @xs_Mod to i32**
call void @new(i32** %reg2, i32 3)
%reg3 = load %List_Mod** @xs_Mod
%reg4 = bitcast %List_Mod* %reg3 to %_Cons_Mod*
%reg5 = mul i32 5, 1
%reg6 = getelementptr [10 x i32]* @__GC___Cons_Mod, i32 0, i32 %reg5
%reg7 = getelementptr %_Cons_Mod* %reg4, i32 0, i32 0
store i32* %reg6, i32** %reg7
```

Listing 4.10: Allocation of a List and assigning GCINFO.

Row 5 prepares the offset used to point at the second sub array of `_GC___Cons_Mod`. This pointer is then stored in the GCINFO field of the newly allocated List.

4.5 Representation of term bindings

Beside type declarations, Kindle modules also contain term bindings. There are two types of term bindings, either named values of atomic type or named functions.

```
data Bind = Val AType Exp
          | Fun [Name] AType ATEnv Cmd
          deriving (Eq,Show)
```

4.5.1 Value bindings

Kindle value bindings are represented as global virtual registers holding a pointer to a heap allocated object. Initialization of the registers are made in the function that initiates the module (see section 4.6).

4.5.2 Function bindings

Before generating a LLVM representation of the function body, the variable environment is updated with the arguments that was provided to the function. These arguments together with global variables from the current module and global variables from imported modules form the variable environment that is used during code generation of the function body. The body

itself consists of a command that computes the desired result, possibly with side effects.

Commands

Kindle commands are translated into LLVM using the function `k2llvmCmd` that takes a `Cmd` node in the Kindle AST and translates that node into the abstract LLVM representation introduced in section 4.2.4. The abstract representation is added to the list of commands that already have been processed.

Variable binding The command `CBind` introduces new variables in the current function. These variables along with their type and associated virtual register are stored in the variable environment used by the current function. There are two different kinds of variable binds in Kindle, those which are heap allocated and those which are stack allocated. Stack allocated variables are allocated only using the LLVM instruction `alloca`, that allocates a certain amount of space on the stack. Heap allocation on the other side needs special treatment. The heap in Timber is managed by the run-time system and to prevent concurrent threads allocating the same memory, all allocation on the heap is done with the function `new` which uses compare-and-swap (CAS) to make sure that the allocations are performed atomically. In the C back-end heap allocation is made with a macro `NEW` but since there is no possibility to use this C macro in the LLVM back-end, the functionality of the macro was moved to a function. The resulting function has the type

```
void @new(i32**, i32)
```

where the first argument is a pointer to an address which will be used to store the address to the memory allocated by the function. The second argument is the amount of memory that should be allocated, measured in whole 32 bit words. To illustrate the use of `new` an example is taken from the function map in the file "Prelude.t".

```
%reg17 = alloca %CONS*
%reg18 = bitcast %CONS** %reg17 to i32**
call void @new(i32** %reg18, i32 3)
```

This sequence allocates memory on the heap that is used to store a cons cell. First a pointer to a cons cell is allocated on the stack, this pointer is cast into an address that is passed as an argument to `new` along with the size of a cons cell in 32 bit words. The function allocates three 32 bit words on the heap and stores the address to these words on the address that was given an argument.

Switch All parts in Timber that has some kind of case analysis such as conditional statement, case statement and guarded equations get transformed into Kindle switch commands. The following representation is used in the Kindle AST.

```
CSwitch Exp [Alt]
```

where `Exp` is the expression that is analyzed and `[Alt]` is the alternatives. The alternatives has the type

```
data Alt = ACon  Name [Name] ATEnv Cmd
         | ALit  Lit Cmd
         | AWild Cmd
         deriving (Eq,Show)
```

The `ALit` construct holds literals that are compared against the expression `Exp` and the command that is to be executed if the comparison succeeds. `AWild` corresponds to the default alternative and holds the command that will be executed if none of the comparisons succeeded. Basically there are three different types of switches that need to be handled in the LLVM back-end, `String`, `Float` and `Int`. In the C back-end, `Int` switches are implemented using switch statements and `Float` and `String` are implemented with if-then-else statements. In the LLVM back-end all three are implemented in the same way using a LLVM translation of the if-then-else statement. The only thing that is different is the method of comparison. `Int` are compared using the instruction `icmp` with condition code `eq` that test for equality, `Float` are compared using `fcmp` with condition code `ueq` that test that the two operands are ordered[1] and equal.

Other The other commands that are a part of kindle; return a value (CRet), run a function for its side effects (CRun), update a variable (CUpd), update a mutable structure object (CUpdS), update an array element (CUpdA), sequential evaluation of two commands (CSeq) and raising an exception (CRaise) are all straightforward translated into a LLVM representation.

Expressions

Except from the different function calls in Kindle, the translation from Kindle expressions into the corresponding LLVM representation is straightforward.

[1] Neither operand is a *quiet NaN*.

Function call There are two major forms of function calls in Kindle. The first is the ordinary call to a local or a global function (ECall) and the second is calling functions that is a part of a structure type (EEnter). As an example we have a function `abs :: Int -> Int`, that takes an integer value and returns the absolute value of the integer. An ordinary call to this function are compiled into a single line of LLVM assembly.

```
%reg1 = call i32 @abs_Mod(i32 10)
```

If instead the function was a part of a Timber struct

```
struct Abs where
    abs :: Int -> Int
```

a call of the function will generate a sequence of LLVM instructions

```
%reg2 = load %Abs_Mod** %reg0
%reg3 = getelementptr %Abs_Mod* %reg2 , i32 0, i32 1
%reg4 = load i32 (%Abs_Mod*, i32)** %reg3
%reg5 = load %Abs_Mod** %reg0
%reg6 = call i32 (%Abs_Mod*, i32)* %reg4(%Abs_Mod* %reg5, i32 5)
```

Line 1 loads the object which of the function `abs` is a part. Line 2 and 3 loads the function itself and line 5 performs the actual call. If the function is passed as an argument the function call is performed using a closure (see section 4.4.1) and the instruction sequence becomes

```
%reg2 = load %CLOS1** %reg0
%reg3 = getelementptr %CLOS1* %reg2 , i32 0, i32 1
%reg4 = load i32* (%CLOS1*, i32*)** %reg3
%reg5 = load %CLOS1** %reg0
%reg6 = inttoptr i32 5 to i32*
%reg7 = call i32* (%CLOS1*, i32*)* %reg4(%CLOS1* %reg5, i32* %reg6)
%reg8 = ptrtoint i32* %reg7 to i32
```

Line 1 loads an object of the closure structure type, line 2 and 3 loads the closure, line 5 casts the 32 bit integer literal into a pointer, that is because the closure expects an argument of type `POLY`. Line 6 performs the actual call to the closure and line 7 casts the result of the closure call back to a 32 bit integer.

Other The other expression that are defined in Kindle are; variables (EVar), literals (ELit), structure field select (ESel), type conversation (ECast) and heap memory allocation (ENew) are all straightforward translated into their LLVM equivalent. More information on heap memory allocation can be found in section 4.5.2.

4.6 Module init-function

The final step of generating the LLVM representation of a Kindle module is to generate a function that initializes the module. The initialization is performed in two steps. In the first step all imported modules are initialized by calling the initialization function in respective module. Step two is to initialize all global virtual registers, by first allocating memory on the heap using the primitive function `new` and then set the global register to point on the allocated memory chunk. Below in listing 4.11 is an example of an init-function, the example is taken from the library module BitOps in the file "BitOps.t".

```
@eqBits8_BitOps = common global %Eq_Prelude* null
```

Definition of a global virtual register that is later initiated in the module initiator function.

```
define void @_init_BitOps() {
    %reg0 = load i1* @INITIALIZED
    %reg1 = icmp eq i1 0, %reg0
    br i1 %reg1, label %label0, label %label1
  label0:
    call void @_init_Prelude()
    ...
    %reg126 = bitcast %Eq_Prelude** @eqBits8_BitOps to i32**
    call void @new(i32** %reg126, i32 3)
    %reg127 = load %Eq_Prelude** @eqBits8_BitOps
    %reg128 = getelementptr [0 x i32]* @__GC__Eq_Prelude , i32 0, i32 0
    %reg129 = getelementptr %Eq_Prelude* %reg127 , i32 0, i32 0
    store i32* %reg128 , i32** %reg129
    ...
    store i1 1 , i1* @INITIALIZED
    br label %label1
  label1:
    ret void
}
```

Listing 4.11: Initiate function in BitOps module.

The `_init_` function follows the same pattern in all modules.

1. The first part above `label0` defines a global Boolean valued virtual register `INITIALIZED` that holds information whether the module already has been initiated or not, this is to prevent the same module from being initialized multiple times.

2. For each imported module the respective module initiator function is called. The BitOps module imports the module Prelude and therefore `_init_Prelude()` is called.

3. All global virtual registers are initialized.

4. Global virtual register `INITIALIZED` is set to true to prevent the module from being initialized again (if the module is imported from several modules).

4.7 Example: compiling map

As an example of how the compilation of a Timber function into LLVM representation is made, we take a closer look on the compilation of the function `map`, which is defined in the library module Prelude in the file "Prelude.t".

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

Listing 4.12: Timber definition of function map

The function `map` takes a polymorphic function `f` of type `a -> b` and maps `f` recursively over a list of type `[a]`. The result is a list of type `[b]`.

4.7.1 Definition of map in Kindle

Before the LLVM back-end generates a LLVM representation of the `map` function, the code is transformed in the front-end of the compiler and translated into the main back-end intermediate language, Kindle (see section 4.2.1).

```
LIST map (BITS32 a_328, CLOS1 a_12, LIST a_13) {
  CONS x_326 = new CONS {
    GCINFO = CONS(primIntTimes(5,
                  primSHIFTR32(
                  primAND32(a_328, (BITS32)2), 1))),
    a = (POLY)0,
    b = (LIST)0
  };
  CONS a_327 = x_326;
    while ((Bool)1) {
        switch ((WORD)a_13) {
          0: { a_327->b = (LIST)0;
               return x_326->b;
             }
          default: { POLY v_194 = ((CONS)a_13)->a;
                     LIST v_195 = ((CONS)a_13)->b;
                     CONS x_329 = new CONS {
                          GCINFO = CONS(primIntTimes(5,
                                        primSHIFTR32(
                                        primAND32(a_328, (BITS32)2), 1))),
```

```
                        a = a_12->Code(v_194),
                        b = (LIST)0
                    };
                    a_327->b = (LIST)x_329;
                    a_327 = (CONS)a_327->b;
                    a_13 = v_195;
                    continue;
                }
        }
    }
    RAISE(1);
}
```

Listing 4.13: Kindle definition of function map.

4.7.2 Definition of map in LLVM

The original function took two arguments, a function of type `a -> b`, an a list with elements of type `a`. When polymorphic functions is translated into LLVM, the function receives an extra argument. The extra argument is a tag, indicating whether the type variables are pointers or scalars.

```
define %LIST* @map_Prelude(i32 %a_328, %CLOS1* %a_12, %LIST* %a_13) {
```

In the case of the function `map`, the extra argument is `i32 %a_328`. The second argument is the function `f` transformed into a unary closure, the third argument is the list.

```
%reg39 = alloca %CONS*
%reg38 = alloca %LIST*
%reg33 = alloca i32*
%reg19 = alloca %CONS*
%reg3 = alloca %CONS*
%reg2 = alloca %LIST*
%reg1 = alloca %CLOS1*
%reg0 = alloca i32
store i32 %a_328 , i32* %reg0
store %CLOS1* %a_12 , %CLOS1** %reg1
store %LIST* %a_13 , %LIST** %reg2
```

All functions that are generated with LLVM starts with allocating memory for all variables on the stack. The main reasons for that is to prevent stack allocations from happen inside a loop, and therefore lead to stack overflow and to be able to take advantage of the LLVM optimization pass `mem2reg`, that promotes memory to registers.

```
%reg4 = bitcast %CONS** %reg3 to i32**
call void @new(i32** %reg4, i32 3)
```

```
%reg5 = load %CONS** %reg3
%reg6 = load i32* %reg0
%reg7 = and i32 %reg6, 2
%reg8 = lshr i32 %reg7, 1
%reg9 = mul i32 5, %reg8
%reg10 = getelementptr [0 x i32]* @__GC__CONS , i32 0, i32 %reg9
%reg11 = getelementptr %CONS* %reg5 , i32 0, i32 0
store i32* %reg10 , i32** %reg11
```

In the third part of the LLVM code an initial cons-cell is allocated on the heap. The first field in the structure type that describes the cons-cell is a pointer to GCINFO. The offset of the sub-array of the array `__GC__CONS` is calculated by a sequence of arithmetic operations. The result from the calculation is used in as argument to the instruction `getelementptr` to index the correct sub-array. A pointer to the sub-array holding the correct GCINFO is stored in the GCINFO field.

```
%reg12 = load %CONS** %reg3
%reg13 = inttoptr i32 0 to i32*
%reg14 = getelementptr %CONS* %reg12 , i32 0, i32 1
store i32* %reg13 , i32** %reg14
%reg15 = load %CONS** %reg3
%reg16 = inttoptr i32 0 to %LIST*
%reg17 = getelementptr %CONS* %reg15 , i32 0, i32 2
store %LIST* %reg16 , %LIST** %reg17
%reg18 = load %CONS** %reg3
store %CONS* %reg18 , %CONS** %reg19
br label %label0
```

Both the element of the cell and the pointer to the next cons-cell is set to `0`. At the end of the code block there is an unconditional break to `label0`, which is the basic block that contains the test expression of the while loop.

```
label1:
  %reg20 = load %LIST** %reg2
  %reg21 = ptrtoint %LIST* %reg20 to i32
  %reg22 = icmp eq i32 %reg21, 0
  br i1 %reg22, label %label4, label %label5
```

This basic-block contains the first test expression in the switch statement. Before the test, the pointer to the next cons-cell is cast to a 32 bit integer. This integer is compared against `0`, where `0` should be interpreted as the empty list. If the test succeeds (the current cell is the last in the list) the control flow moves to `label4`, if there is a next cons-cell the control flow moves to `label5`.

```
label4:
  %reg23 = load %CONS** %reg19
  %reg24 = inttoptr i32 0 to %LIST*
```

```
%reg25 = getelementptr %CONS* %reg23 , i32 0, i32 2
store %LIST* %reg24 , %LIST** %reg25
%reg26 = load %CONS** %reg3
%reg27 = getelementptr %CONS* %reg26 , i32 0, i32 2
%reg28 = load %LIST** %reg27
ret %LIST* %reg28
br label %label5
```

When the pointer to the next cons-cell is null, the whole list has been processed. The pointer to the next cons-cell in the new list is set to null and head of the new list is loaded and returned.

```
label5:
  %reg29 = load %LIST** %reg2
  %reg30 = bitcast %LIST* %reg29 to %CONS*
  %reg31 = getelementptr %CONS* %reg30 , i32 0, i32 1
  %reg32 = load i32** %reg31
  store i32* %reg32 , i32** %reg33
  %reg34 = load %LIST** %reg2
  %reg35 = bitcast %LIST* %reg34 to %CONS*
  %reg36 = getelementptr %CONS* %reg35 , i32 0, i32 2
  %reg37 = load %LIST** %reg36
  store %LIST* %reg37 , %LIST** %reg38
```

The next basic-block handles the case where the cell has a pointer to another cons-cell. First the data of the current cell and the pointer to the next cons-cell loaded.

```
%reg40 = bitcast %CONS** %reg39 to i32**
call void @new(i32** %reg40, i32 3)
%reg41 = load %CONS** %reg39
%reg42 = load i32* %reg0
%reg43 = and i32 %reg42, 2
%reg44 = lshr i32 %reg43, 1
%reg45 = mul i32 5, %reg44
%reg46 = getelementptr [0 x i32]* @__GC__CONS , i32 0, i32 %reg45
%reg47 = getelementptr %CONS* %reg41 , i32 0, i32 0
store i32* %reg46 , i32** %reg47
```

Then the a new cons-cell is allocated and the GCINFO field is initiated.

```
%reg48 = load %CONS** %reg39
%reg49 = load %CLOS1** %reg1
%reg50 = getelementptr %CLOS1* %reg49 , i32 0, i32 1
%reg51 = load i32* (%CLOS1*, i32*)** %reg50
%reg52 = load %CLOS1** %reg1
%reg53 = load i32** %reg33
%reg54 = call i32* (%CLOS1*, i32*)* %reg51(%CLOS1* %reg52, i32* %reg53)
%reg55 = getelementptr %CONS* %reg48 , i32 0, i32 1
store i32* %reg54 , i32** %reg55
```

The closure is applied to the element of the data field in the old cell and the result is stored in the data field of the new cell.

```
    %reg56 = load %CONS** %reg39
    %reg57 = inttoptr i32 0 to %LIST*
    %reg58 = getelementptr %CONS* %reg56 , i32 0, i32 2
    store %LIST* %reg57 , %LIST** %reg58
```

The pointer to the next cell in the new list is set to null.

```
    %reg59 = load %CONS** %reg19
    %reg60 = load %CONS** %reg39
    %reg61 = bitcast %CONS* %reg60 to %LIST*
    %reg62 = getelementptr %CONS* %reg59 , i32 0, i32 2
    store %LIST* %reg61 , %LIST** %reg62
    %reg63 = load %CONS** %reg19
    %reg64 = getelementptr %CONS* %reg63 , i32 0, i32 2
    %reg65 = load %LIST** %reg64
    %reg66 = bitcast %LIST* %reg65 to %CONS*
    store %CONS* %reg66 , %CONS** %reg19
    %reg67 = load %LIST** %reg38
    store %LIST* %reg67 , %LIST** %reg2
    br label %label0
    br label %label3
```

In the last part of the basic-block, the variables used in the body of the while loop is prepared for the next iteration and the control flow moves to `label0`.

```
  label3:
    br label %label0
```

This is only an effect of the compilation schema of the switch statement and the basic-block is never entered.

```
  label0:
    %reg68 = icmp eq i1 1, 1
    br i1 %reg68, label %label1, label %label2
```

The main work in the `map` function is done inside the infinite while-loop, the test expression only compares `1` and `1` for equality, which is always true, and the control flow moves to `label1`.

```
  label2:
    call void @RAISE(i32 1)
    unreachable
}
```

The last basic-block, which is outside of the infinite while-loop and the control flow should never enter. The basic-block only contains a call to the function `RAISE` that raises an exception and an instruction `unreachable`. `unreachable` is added because the LLVM assembler does not permit a function call as the last instruction in a basic-block.

Chapter 5

Evaluation

In this chapter the LLVM back-end is compared against the C back-end. The comparison consists of a small set of Timber programs that are compiled with respective back-end and the performance of the resulting executables are compared against each other. All programs were run on a Core 2 Duo P8600 processor running at 2.40GHz, with 2048MB of memory under Ubuntu 10.04 (2.6.32-22-generic kernel). The execution time given is the mean value of 20 subsequent executions. The source code for each program can be found in the Appendix.

5.1 Performance

As seen in table 5.1, the LLVM back-end produces code of significantly higher quality than the C back-end on all programs except two, and in those cases the LLVM back-end is not far behind. One interesting result is the last row, a program that calculates the 45th Fibonacci number, with the 45 given as a constant inside the program. in this case, LLVM really takes advantage of this knowledge and produces code that executes approximately twelve times faster than the code compiled with the C compiler.

5.2 File size

Of the programs that were compiled, the LLVM beck-end generated intermediate files that were smaller than the intermediate files generated by the C back-end, in five out of eight cases. But when comparing the size of the executable file, the LLVM back-end produced smaller files in all cases.

Program	**C [s]**	**LLVM [s]**	**Relative**
Tak (`Int`)	3.6756	2.1874	1.6804
Tak (`Float`)	4.3028	2.2697	1.8958
Tak (Polymorphic)	10.4161	9.3560	1.1133
Ackermann (`Int`)	2.4364	1.1835	2.0586
Ackermann (`Float`)	4.8131	1.9042	2.5276
Ackermann (Polymorphic)	26.4403	18.7475	1.4103
Primes	1.2801	1.3065	0.9798
Concurrent primes	22.7399	14.7495	1.5417
Mandelbrot set	2.5975	2.6340	0.9861
Matrix multiplication (`Int`)	2.5547	1.9097	1.3377
Matrix multiplication (`Float`)	2.8432	2.0688	1.3743
Fibonacci (argument)	19.6203	11.2080	1.7506
Fibonacci (constant)	19.6191	1.6290	12.0436

Table 5.1: Comparison of the execution time.

	Executable		Intermediate	
Program	**LLVM**	**C**	**LLVM (.bc)**	**C (.o)**
Tak function	50541	74956	1796	2524
Ackermann function	50541	74956	2356	2652
Primes	50607	75057	2716	2468
Concurrent primes	56426	86049	10488	9200
Mandelbrot set	62617	86847	8936	8284
Matrix multiplication	51226	75769	4356	4640
Fibonacci argument	50480	74905	1084	1584
Fibonacci constant	50480	74905	1080	1476

Table 5.2: Comparison of file size measured in bytes.

Chapter 6

Conclusion

This thesis has described the design and implementation of the LLVM back-end of the Timber compiler. Looking at the implementation, no significant modifications on the run-time system was necessary in order to make it work with the LLVM back-end. The LLVM back-end is implemented as a separate module and none of the existing modules in the compiler was altered.

Compared to the C back-end, the LLVM back-end is larger (measured in lines of code), this is mainly because the LLVM back-end uses an intermediate abstract representation of the LLVM assembly language, which is pretty-printed to a text file, while the C back-end uses a pretty-printer directly, without an abstract representation. However, the use of an abstract representation can be eliminated by using Haskell FFI bindings to the LLVM API (see 6.1.1).

As seen in chapter 5, the LLVM back-end performs nearly as good as the C back-end or better. In the two cases where the C back-end was the fastest; on the Mandelbrot test and on the single threaded Primes test, the LLVM back-end performed nearly as good, but in the rest of the test cases the LLVM back-end performed significantly better than the C back-end. This is a very promising result given that the development of the LLVM back-end has only started.

Further possibilities with LLVM is to utilize the full optimization facilities that are offered. Compile-time optimization of single modules generated from single Timber modules, link-time optimization with whole program scope and optimizations on the linked program. The C source code generated from the original C back-end has a distinct style and so does the LLVM assembly code generated. This should imply that there is set of LLVM optimization passes that could be extra beneficial for the certain style of assembly code that the LLVM back-end generates. This could lead to smaller size of generated files or even faster execution time. In the optimal case, the optimizations will

produce files that are both smaller in size and have shorter execution time.

Another possibility with LLVM is to use the LLVM optimization pass framework and write optimization passes that are especially targeted to the certain style of LLVM assembly code that the Timber compiler produces.

The conclusion is that the LLVM compiler infrastructure offers a suitable back-end for the Timber compiler and it is capable of generating code that has higher quality than code generated by the C back-end, both measured in size and performance.

6.1 Further work

The further work can be divided into two parts, one focuses on the modifying the LLVM back-end of the Timber compiler and the other focuses on utilize the LLVM optimization machinery.

6.1.1 Using Haskell FFI bindings to LLVM

Instead of generate LLVM assembly code from the Timber compiler, the code generator should use Haskell FFI bindings to the LLVM API. Those bindings should operate on a lower than those that currently are available[1]. This would reduce the complexity of the LLVM back-end and enable writing LLVM bitcode files directly instead of writing LLVM assembly code that are assembled with a command line tool.

6.1.2 Optimizations

The implemented LLVM back-end only make use of the standard optimization passes, it would be interesting to make use of the full optimization machinery that LLVM provides. This involves both extending the Timber compiler with support for flags that is passed to the LLVM optimizer and to perform an investigation of which passes that are suitable for the code that are generated by the Timber LLVM back-end.

Bibliography

[1] Lennart Augustsson and Bryan O'Sullivan. *HackageDB: llvm-0.7.1.1*. Mar. 2010. URL: `http://hackage.haskell.org/package/llvm`.

[2] Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. *The Semantic Layers of Timber*. Vol. 2895. 2003, pp. 339–356.

[3] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004. ISBN: 155860698.

[4] The IEEE and The Open Group. *The Open Group Base Specifications Issue 7*. Mar. 2010. URL: `http://www.opengroup.org/onlinepubs/9699919799/`.

[5] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". *See* `http://llvm.cs.uiuc.edu`. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[6] Chris Lattner. *The LLVM Compiler Infrastructure*. Feb. 2010. URL: `http://llvm.org/`.

[7] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California 2004.

[8] Johan Nordlander. "Reactive Objects and Functional Programming". PhD thesis. Chalmers University of Technology, 1999.

[9] John van Schie. "Compiling Haskell to LLVM". MA thesis. Utrecht University, 2008.

[10] Reid Spencer. *The Often Misunderstood GEP Instruction*. Feb. 2010. URL: `http://llvm.org/releases/2.6/docs/GetElementPtr.html`.

[11] David Anthony Terei. "Low Level Virtual Machine for Glasgow Haskell Compiler". MA thesis. The University of New South Wales, 2009.

[12] Unknown. *LLVM: Frequently Asked Questions*. Mar. 2010. URL: `http://llvm.org/docs/FAQ.html`.

Appendix A

Source code

A.1 Tak function

Calculates the Tak function.

```
-- Integer arguments
takI :: Int -> Int -> Int -> Int
takI x y z
    | x <= y    = y
    | otherwise = takI (takI (x-1) y z) (takI (y-1) z x) (takI (z-1) x y)

-- Floating point arguments
tFloat :: Float -> Float -> Float -> Float
tFloat x y z
    | x <= y    = y
    | otherwise = takF (takF (x-1) y z) (takF (y-1) z x) (takF (z-1) x y)

-- Polymorphic arguments
takP x y z
    | x <= y    = y
    | otherwise = takP (takP (x-1) y z) (takP (y-1) z x) (takP (z-1) x y)
```

A.2 Ackermann function

Calculates the Ackermann function.

```
-- Integer arguments
ackI :: Int -> Int -> Int
ackI m n
    | m == 0          = n+1
    | m > 0 && n == 0 = ackI (m-1) 1
    | otherwise       = ackI (m-1) (ackI m (n-1))
```

```
-- Integer arguments
ackF :: Float -> Float -> Float
ackF m n
    | m == 0          = n+1
    | m > 0 && n == 0 = ackF (m-1) 1
    | otherwise       = ackF (m-1) (ackF m (n-1))

-- Polymorphic arguments
ackP m n
    | m == 0          = n+1
    | m > 0 && n == 0 = ackP (m-1) 1
    | otherwise       = ackP (m-1) (ackP m (n-1))
```

A.3 Number of primes

Calculates the number of prime numbers below or equal to a given integer.

```
module Primes where

import POSIX

root :: World -> Cmd () ()
root w = do
   env = new posix w

   limit :: Int
   limit = fromRight (parse (env.argv!1))

   primesCounter = new primes limit
   ct <- primesCounter
   env.stdout.write (show ct++"\n")
   env.exit 0

primes limit = class
   primesBound = limit `div` log3 limit

   primes := uniarray primesBound 0
   count  := 0

   isPrime k = loop 0
      where loop n = do
              p = primes!n
              if p*p > k then
                 result True
              elsif k `mod` p  == 0 then
                 result False
              else
                 r <- loop (n+1)
```

```
                result r

   checkFrom k = do
     p <- isPrime k
     if p then
        primes!count := k
        count := count + 1
     if k < limit then checkFrom (k+1)

   result request
     primes!0 := 2
     count := 1
     checkFrom 3
     result count

log3 :: Int -> Int
log3 n
  | n < 3     = 0
  | otherwise = 1 + log3 (n `div` 3)
```

A.4 Concurrent number of primes

Calculates the number of prime numbers below or equal to a given integer using a concurrent approach.

```
module ConcPrimes where

import ConcMap
import POSIX
import Data.Functional.List -- sum

-- Computes and prints the number of primes smaller than or equal to
-- command line argument.
root :: World -> Cmd () ()
root w = do
    env = new posix w

    limit = fromRight (parse (env.argv!1))
    limr  = isqrt limit -- floor (sqrt (fromInt limit))
    smallPrimes = primesTo limr
    finish ps = action
        count = length smallPrimes + sum ps
        env.stdout.write (show count ++ "\n")
        env.exit 0

    tasks = ((limr-1)*limr+1,limit) : [((j-1) * limr + 1, j * limr)
                                       | j <- [2..limr-1]]
```

```
                                    ]

    cmap (countPrimes smallPrimes) tasks finish

-- counts primes in interval (from,to);
-- requires ps to be list of all primes up to at least sqrt(to).
countPrimes ps (from,to)  = loop from 0
  where loop k ack
          | k > to        = ack
          | noFactors ps  = loop (k+1) (ack+1)
          | otherwise     = loop (k+1) ack
          where
            noFactors []       = True
            noFactors (p : ps) = p*p > k ||
                                 (k `mod` p /= 0 && noFactors ps)

-- Used to compute list of "small" primes for use as argument
-- to countPrimes.
primesTo n             = sieve [2..n]
   where sieve (p:ps)
           | p*p > n   = p : ps
           | otherwise = p : sieve [ q | q <- ps, q `mod` p /= 0 ]

isqrt n
  | n < 4 = 1
  | otherwise = isqrt' n

isqrt' n
  | (n1 + 1) * (n1 + 1) <= n = n1  + 1
  | otherwise = n1
  where n1 = 2 * (isqrt (n `div` 4))
```

A.4.1 Concurrent map

```
module ConcMap where

  cmap :: (a -> b) -> [a] -> ([b] -> Action) -> Cmd _ ()
  cmap f xs cb = do
       c = new cmap1 f xs cb
       c
       result ()

private

  cmap1 :: (a -> b) -> [a] -> ([b] -> Action) -> Class _
  cmap1 f xs cb = class
      numRunning := 0
      results := array []
```

```
      storeResult x v = request
        if size results == 0 then
           results := uniarray (length xs) v

        results!x := v
        numRunning := numRunning - 1
        if numRunning == 0 then
           cb (fromArray results)
        result ()

      launchTasks = action
         forall x <- xs do
           n = numRunning
           ev = new class result action (storeResult n (f x))
           ev
           numRunning := numRunning + 1

      result launchTasks

  fromArray :: Array a -> [a]
  fromArray a = fromArray' [] (size a-1)
    where fromArray' xs n
            | n < 0 = xs
            | True  = fromArray' (a ! n : xs) (n-1)
```

A.5 Mandelbrot set

Generates an square image of the Mandelbrot set.

```
module MandelT where

import BitOps
import ConcMap
import POSIX

root :: World -> Cmd () ()
root w = do
    env = new posix w
    wid = fromRight (parse (env.argv ! 1))  -- width of image in pixels

    finish res = action
       env.stdout.write ("P4\n" ++ show wid ++ " " ++ show wid ++ "\n")
       forall i <- res do
           env.stdout.write i
       env.exit 0
    cmap (mk_row wid) [0..wid-1] finish
```

```
mk_row w y = loop (wb-1) ""    -- produce image row at y
  where
   wb = w `div` 8              -- width of image in bytes
   wc = 2 / fromInt w          -- width of a pixel in the complex plane
   ci = fromInt y * wc - 1

   loop j ack
      | j < 0 = ack
      | otherwise = loop (j-1) (toEnum (toInt (mk_byte (j*8) 8 0)) : ack)

   mk_byte x n b
     | x < w = if n == 0
               then b
               else mk_byte (x+1) (n-1) ((b .<<. 1) .|. v)
     | otherwise = b .<<. n
    where cr = fromInt x * wc - 1.5
          v = fractal 0 0 50
          fractal :: _ -> _ -> Int -> BITS8
          fractal r i k
            | r2 + i2 > 4 = 0
            | k == 0      = 1
            | otherwise   = fractal (r2-i2+cr) ((r+r)*i+ci) (k-1)
            where (r2,i2) = (r*r,i*i)
```

A.6 Matrix multiplication

Multiplication of two matrices.

```
matmult a b = do
        res = new class
                m = size a
                n = size b
                p = size (b!0)
                r := uniarray m (uniarray p 0)
                result request
                       forall i <- [0..m-1] do
                              forall j <- [0..n-1] do
                                     forall k <- [0..p-1] do
                                            r!i!j := r!i!j + a!i!k * b!k!j
                       result r
        r <- res
        result r
```

A.7 Fibonacci

Calculates the nth number in the Fibonacci sequence.

```
fib :: Int -> Int
fib n = if n <= 1 then 1 else fib (n-1) + fib (n-2)
```