Low Level Virtual Machine for Glasgow Haskell Compiler

By
David Anthony Terei

Supervisor
Manuel M. T. Chakravarty

THESIS

Submitted in partial fulfilment of the requirements for the degree of
Bachelor of Science in Computer Science

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

Computer Science and Engineering,
The University of New South Wales.

20th of October, 2009

Abstract

This thesis details the motivation, design and implementation of a new back-end for the Glasgow Haskell Compiler which uses the Low Level Virtual Machine compiler infrastructure for code generation.Haskell as implemented by GHC was found to map remarkably well onto the LLVM Assembly language, although some new approaches were required. The most notable of these being the use of a custom calling convention in order to implement GHC's optimisation feature of pinning STG virtual registers to hardware registers. In the evaluation of the LLVM back-end in regards to GHC's C and native code generator back-end, the LLVM back-end was found to offer comparable results in regards to performance in most situations with the surprising finding that LLVM's optimisations didn't offer any improvement to the run-time of the generated code. The complexity of the LLVM back-end proved to be far simpler though then either the native code generator or C back-ends and as such it offers a compelling primary back-end target for GHC.

Acknowledgements

I would like to first and foremost thank my parents. My mother Ann Terei and my father, Tom Terei. They have always been beside me. I would also like to thank my supervisor Manuel Chakravarty, his knowledge and answers have been invaluable. Finally I would like to thank all my friends, perhaps one of them will read this.

Contents

List of Tables

List of Figures

Listings

1

Introduction

Compilers comprise some of the largest and most complex software systems, dealing with the parsing, transformation and eventual generation of executable machine code for a program. They are usually designed, conceptually at least, in two components, a *front-end*, that deals with the parsing of the language into an in memory representation and a variety of transformations; and a *back-end*, which handles the generation of executable code. A great amount of research has been done into the design and implementation of back-ends, particularly in trying to create a 'universal' back-end, one which can efficiently support a wide variety of languages. Language designers have a choice in the implementation of their compiler of how the code should be eventually executed, that is, of which type of back-end to target. This decision is usually made based on the level of control the language designer wants over the generated executable code, the more control that is desired then the greater the amount of work needed from them.

In this thesis I will be looking at the generation of executable code for for the Haskell programming language. I will be evaluating an implementation of a new back-end target for the Glasgow Haskell Compiler (*GHC*) [10], a cutting edge, industrial strength compiler for the Haskell programming language. Firstly though we will look at the approaches compiler writers usually take for code generation.

1.1 Typical Back-end Targets

The generation of executable code can be done through the use of many different types of back-ends but they can usually be classed as one of the following [32]:

- Assembly
- High Level Languages
- Virtual Machines
- High Level Assembly

The choice between them is a trade off between several factors such as the level of control, performance, and the amount of work which can be leveraged from others.

Producing machine code through the generation of assembly is the traditional path and offers complete control to the compiler writer, allowing theoretically for the most efficient code due to its lack of overhead and flexibility. However the generation of assembly is no trivial task, requiring a considerable investment of time and also has the disadvantage of being specific to a particular machine architecture and operating system. Issues such generating the position independent code needed for shared libraries and efficient register allocation only increase this complexity, making it difficult to implement and harder still to optimise and maintain. This choice usually involves the largest amount of work as well since none of it is being outsourced.

Because of this generating executable code through targeting a high level language has become a very popular choice [16,30] due to its relative ease compared to assembly and ability to support multiple platforms. The high level language most often used is C, due to its level nature and the availability of high quality C compilers on most platforms. The actual generation of the machine code is left to the high level language compiler. This approach is not without its own disadvantages, such as the loss of control over the details of code generation, incurring a performance overhead. The lack of tail call support in C is also a significant problem for a language such as Haskell which like most functional language uses tail recursion extensively.

Another approach which has risen greatly in popularity in recent years is the targeting of virtual machines, such as the Java Virtual Machine (*JVM*) [25] or Common Language Runtime (*CLR*) [18]. These virtual machine provide rich environment, with a number of readily available garbage collectors, exception handling and direct access to huge programming libraries. These rich features and the portability they give to programs make them an attractive target. However they usually have the worst performance compared to the other options, particularly when the garbage collector and other design choices made by the virtual machine don't correspond well

to the programming language being compiled. For example functional languages like Haskell tend to allocate far more frequently and aggressively then procedural languages, a case which virtual machine implementations aren't usually optimised for [21].

The final option that compiler writers can target is a high level assembly language. These languages provide a level of abstraction between assembly and a high level language, usually being designed as a portable assembly language, abstracting away issues such as calling conventions and hardware registers. They provide one of the lowest levels of abstraction generally possible while still being platform independent. Unlike high level languages or virtual machines such as the JVM, they don't provide features like garbage collection or exception handling, leaving the compiler writer in full control to implement them if needed. The Low Level Virtual Machine (*LLVM*) [24] is a compiler infrastructure designed around such a language. It provides a low level portable compilation target, with high quality static compilers. I believe it is the most suitable back-end for the Glasgow Haskell Compiler and as part of this thesis I have implemented a new back-end for GHC which targets LLVM.

1.2 Low Level Virtual Machine Back-end for GHC

The Low Level Virtual Machine is state-of-the-art optimising compiler framework, providing the raw tools needed to build high quality compilers. It provides a high performance static compiler back-end which can be targeted by compiler writers to produce machine code. Started in 2000, it offers a promising new back-end target for many existing compilers including GHC. LLVM is situated at the right level of abstraction for a language like Haskell, imposing no design decisions on GHC, while providing the required features such as efficient tail call support. It will also allow a significant amount of work to be offloaded from the GHC developers. GHC currently offers two approaches to generating executable code, by either generating C, or by producing native assembly. Both of these suffer from the issues outlined above in *section 1.1*. Because of this I believe that LLVM is the most appropriate back-end target for GHC and the Haskell programming language.

1.3 Research Aims

As part of this thesis I have implemented a new Low Level Virtual Machine (*LLVM*) back-end for the Glasgow Haskell Compiler (*GHC*). It is my hypothesis that LLVM will prove to be the most suitable target for GHC. As part of this work I aim to answer the following questions:

- Is the implementation reasonably straight forward. How does it compare in complexity to GHC's other back-ends?

- Is the generated code of similar quality to GHC's other back-ends?
- Which optimisations no longer apply and what new optimisations are now possible with LLVM?

1.4 Organisation of this Thesis

In order to understand the motivation of this thesis, Chapter 2 discuss the current state of GHC. It also equips the reader with the background information needed to understand the technology and material drawn on in the rest of the thesis. Chapter 3 describes the design of the new LLVM back-end for GHC, evaluating the approach taken and the problems encountered in this work. Chapter 4 evaluates the new LLVM back-end in comparison with GHC's pre-existing code generators. This evaluation is done in terms of complexity, performance and flexibility. Chapter 5 concludes the work and outlines possible future work in this area.

2

Background

2.1 Compiler Design

As the design and implementation of compilers is a well understood problem that draws on a considerable amount of theory, a general design pattern for them has emerged which we will explore in this section. A very high level view of a compiler can be broken down into the following three stages, as seen in *figure 2.1* [32]:

1. Parse source language to an intermediate representation (*IR*).
2. Transformations of the intermediate representation.
3. Generation of machine code from IR.

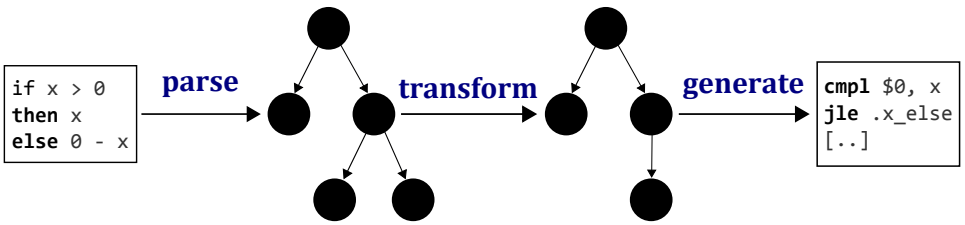


Figure 2.1: Compiler Pipeline

In this thesis I will be focusing on the last step, the generation of machine code.

Ideally we want to design compilers in such a way that we can separate them into two distinct components, the so called *front-end* and *back-end*. The front-end deals with stages 1 and 2, the parsing of the source language to an intermediate representation, such as an abstract syntax tree, and some subsequent transformations of this IR to improve the performance of the code as well as simplifying it to make code generation easier. This usually involves more then one IR, with each transformation into a new IR simplifying the code and representing it at a lower abstraction level, closer to the machine architecture he compiled program will run on. Eventually the compiler will transform the code into an IR which is suitable for code generation and at this point the 3rd stage is invoked, the back-end, generating machine code.

Separating a compiler into these two distinct components allows for the reuse of a single back-end by a wide variety of front-ends which each implement their own language. This is done by each front-end compiling their source language to the IR that the shared back-end accepts, *targeting* it as is said. This allows for a great reduction in the amount of work which needs to be done by the compiler community at large, sharing the common back-end work among all languages rather then having each compiler produce their own back-end and duplicate work.

Despite this advantage there has been little success in trying to achieve this strong separation due to some significant challenges in the design, particularly in regards to the design of the common intermediate representation. The main issue is trying to specify an IR which can efficiently support the wide variety of programming languages which exist. The design of such an IR encounters the following issues:

1. The IR needs to be of a fairly low level so it is able to efficiently support most languages, otherwise the high level abstractions conflict with languages that have significantly different semantics. However a low level representation cannot support as aggressive optimisations as higher level representations due to the loss of knowledge of what the code is trying to achieve [24].

2. Another issue has also been how to efficiently support high level services such as garbage collection in a universal and portable manner in the IR. The problem here is that these services usually require tight co-operation between the back-end and front-end of a compiler to implement, especially with decent performance. Many garbage collection designs for example require walking the stack of the program after it has been suspended during execution. How to allow front-end run time service to perform such operations without exposing target specific low level details is an unsolved problem.

3. There has also been a historical *'chicken and egg'* problem, since compiler writers don't want to invest the considerable amount of resources needed to produce a high quality back-end unless they know it will be used by compiler front-ends, while compiler writers

producing compiler front-ends don't want to make their language dependent on a specific back-end until its reached a mature stage of development.

While there have been a few attempts to solve this problem, two are of notable interest for this thesis, *C–* [21] and *LLVM*.

A reduced and much less ambitious version of C– is actually used by GHC within its compilation pipeline, as will be outlined in *section 2.4.3*. C– tries to solve all of the issues outlined above, using C as its starting point and defining a run time interface for the back-end and front-end to utilise to efficiently implement high level services. It unfortunately though it has so far not managed to gain much traction primarily due to point 3, the *chicken and egg* problem, with a full compiler for the language not existing.

LLVM is of course one of the main focuses of this thesis and also attempts to address the problems outlined above. One of its unique features is its approach to point 1, with LLVM using a fairly low level instruction set but with high level type information to enable aggressive optimisations to be done on the code. LLVM originally though made no attempt to solve point 2 but has recently addressed this using ideas not too dissimilar from those of *C–*, defining a common API that the back-end is aware of but which is implemented by the front-end developer [23]. It also took a significant amount of work by LLVM developers before they reached a stage where other compiler writers began to utilise their work, with LLVM having been in continuous development for over 10 years now, its only in the last couple of years that this occurred. LLVM will be discussed further in *section 2.6*

2.2 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (*GHC*) is a state-of-the-art compiler for the *Haskell* programming language. Started in 1989, it is mostly written in Haskell itself with the runtime system in C. It is primarily developed by Simon Peyton Jones and Simon Marlow of Microsoft Research [10]. This thesis focuses on the current back-end's of GHC, that deal with with the generation of executable code.

GHC currently supports a number of different architectures including x86, x86-64, PowerPC and SPARC, as well as a running on a variety of different operating systems including Linux, Mac OS X and Windows. GHC can currently generate code through two different back-ends, a C code generator and a native code generator (*NCG*) which produces assembly code. As part of the work of this thesis I implemented a third option, an LLVM back-end which uses LLVM for executable code generation. This I believe is the most appropriate code generator for GHC due to significant issues with the C and NCG, as are outlined in *section 2.3.1* and *section 2.3.2* respectively. First though we must at the two primary 'modes' in which GHC can generate executable code, the so called *unregistered* and *registered* modes.

2.3 GHC Compilation Modes

GHC is capable of compiling code in two different modes, *unregistered* and *registered*, with registered mode as the name implies being a superset of unregistered mode in terms of the optimisations applied and work required from the back-end code generators. Unregistered mode is used largely for portability reason and is not a viable code generation option to the loss of performance compared to the registered mode.

Unregistered mode is the first mode GHC supported and is only properly implemented by the C code generator [1], it represents the limit of the performance GHC can achieve using fairly portable techniques to compile GHC to C code [20].

Registered mode is implemented by both the C and NCG back-ends and involves primarily two optimisations which significantly increase the performance of generated code. The first optimisation is known as *TABLES_NEXT_TO_CODE* and optimises the data layout of the code GHC produces, this is further explained in *section 2.5*. The second optimisation though is far more significant, it deals with the register allocation for the generated code. As part of the execution model that GHC uses for Haskell, it defines a variety of virtual registers which mirror many of the registers typically found on a CPU, such as a stack pointer register. These are examined in more detail in *section 2.4.2* but their purpose is to create a virtual machine architecture which can be explicitly managed instead of using the C stack for example. In *unregistered* mode these virtual registers are all stored on the heap, needing memory access for any reads and writes. Given the frequency of access of these virtual registers, this creates a significant amount of memory traffic which becomes a limiting factor on the performance of the generated code. In *registered* mode many of these virtual registers are instead permanently assigned to a specific hardware registers (*register pinning*), greatly improving performance. The runtime of code compiled in *registered* mode as opposed to *unregistered* mode is on average 55% shorter. More details of this can be seen in *section 4*.

Compiling code in *registered* mode as compared to unregistered is a fair challenge as both these optimisations require specifying very low level and architecture specific details that the generated code must conform to. With this is mind we will now investigate GHC's two existing back-ends and evaluate their current state as a code generator for GHC.

2.3.1 C Code Generator

GHC's C back-end was the first code generation path that GHC supported, it provides the advantage of being relatively portable and fast but also has the disadvantages in its lack of

[1] While the NCG can mostly compile code in unregistered mode, there are a few assumptions it currently makes which prevent it from doing so. These could however be easily fixed.

control, complexity and performance. The C back-end is fairly portable, with GHC using the GNU GCC C compiler (*GCC*) [13] for compilation, which is well supported on most POSIX based platforms. The generated code also runs with fairly good performance, enabling Haskell programs to approach the speed of equivalent C programs. However there are quite a few problems with the C back-end, most of them related to optimisations needed to achieve an acceptable level of performance from the generated code. This is especially true of the techniques used to enable the C back-end to generate registered code.

Firstly to handle the pinning of virtual registers in registered mode, it uses a GCC specific feature called *global register variables* [14] which enables C variables to be fixed to a chosen hardware register. This as well as the use of some other GCC extensions ties GHC specifically to GCC, which isn't well supported on some platforms such as Microsoft Windows. The second technique that GHC uses to increase the performance of code generated by the C back-end is to process the assembly code produced by GCC, running a number of optimisation passes over it. One of these passes also enables the optimised data layout used in registered mode, achieving this by manually rearranging the assembly. This creates a dependency for GHC on not only just GCC but also specific versions of GCC due to the differences in assembly each produces. Ongoing work is then required to keep the C back-end working with new versions of GCC. Even with these techniques, there are still a variety of optimisations that the GHC team cannot implement in the C back-end due to it high level nature. While the C back-end functions reasonably well in unregistered mode, the performance degradation is far too great for it to be considered an appropriate code generation option.

Finally the compilation times of the C back-end are also very slow, especially when compared to GHC's other backend, the native code generator. Usually the compilation speed of a typical Haskell program via the C back-end takes on the order of twice as long as compilation via the native code generator.

2.3.2 Native Code Generator

The other back-end which GHC supports is the so called Native Code Generator (*NCG*). This is a full compilation pipeline, with GHC's native code generator producing machine code for the particular platform it is running on. GHC's native code generator shares the usual advantages and disadvantages of a back-end which produces assembly, it can generate the fastest code but requires a considerable amount of work and is specific to a one particular platform. GHC's NCG requires each platform, such as x86, SPARC, and PowerPC to perform their own code generator with only the register allocator being shared among all platforms. It is also quite difficult to produce these code generators so that they produce optimal code, with a large amount of optimisations required and each particular architecture requiring a lot of fine tuning to ensure optimal register and instruction selection. GHC could benefit from a lot of common

optimisations in the native code generator, such as standard loop optimisations like partial redundancy elimination (*PRE*), unrolling, strength reduction, as well as a whole bag of others. However this involves a lot of rather mundane work which is of little interest to GHC developers. These types of optimisations are also something that LLVM brings with it for free, representing a far more attractive avenue.

The NCG can easily support compiling code in registered mode and as such doesn't suffer from the same issues the C code generator. However it has a lot of other problems to handle, such as the generation of position independent code which is required for shared libraries. Compared to the C back-end it offers a slight performance increase and much faster compilation speeds. A more detailed comparison of the two can be found in *section 4*.

2.4 Glasgow Haskell Compiler Design

In this section we will give a brief overview of the design of GHC, focusing on areas which are relevant to the implementation of a Low Level Virtual Machine (*LLVM*) back-end for GHC.

2.4.1 Pipeline Overview

Firstly we begin with a look at the pipeline of GHC, an overview of this can be seen in *figure 2.2* [6, 20]. GHC's pipeline involves the following intermediate representations and stages, beginning with Haskell source code and ending with executable code:

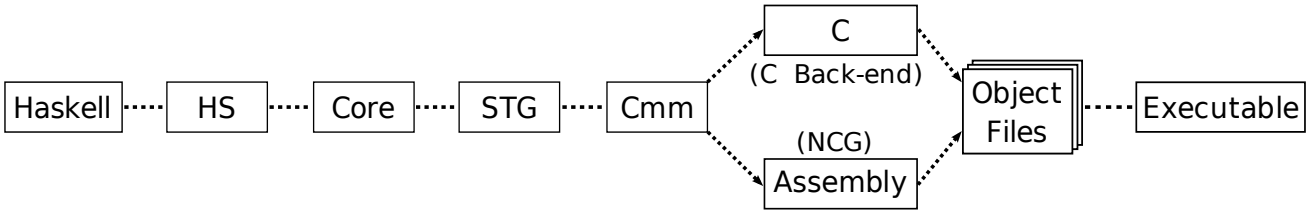


Figure 2.2: GHC Pipeline

- **HS**: An in memory representation of the Haskell programming language with all its syntax represented. Parsing, type-checking and renaming are all done using this representation, or slight variations of.
- **Core**: A version of lambda calculus with some extensions. Designed to be large enough to efficiently express the full range of Haskell and no larger. A variety of simplifications and optimisations are done in this representation.

- **STG**: The *Spineless Tagless G-Machine* [19] is an abstract machine representation of Haskell and makes explicit its execution model. It will be discussed further in *section 2.4.2*. STG is quite similar to Core but more suitable for code generation.
- **Cmm**: Cmm is a variation on the *C–* language [21]. It represents the Haskell program in a procedural form, acting as the basis for all back-end work. It will be discussed further in *section 2.4.3*.

2.4.2 Spineless Tagless G-Machine

The Spineless Tagless G-Machine is a so called *"abstract machine"*, it distils the aspects of Haskell's execution model using well defined *operational* semantics that can be efficiently mapped to stock hardware. One can think of it in a slightly similar fashion to virtual machines such as the Java Virtual Machine [25] but designed explicitly for no-strict, higher-order functional languages such as Haskell.

As part of the operational semantics of STG, it defines a number of virtual registers for manipulating things such as the heap and stack, as well as generic registers for argument passing between function calls. A listing of these virtual registers, with description can be seen in *table 3.1* [11] [2]. These virtual registers are also used by the run-time system (*RTS*) which is mostly written in C, providing an interface between the two components. As discussed in *section 2.3* GHC can handle these virtual registers in two ways, one is to store them all on the *Heap* which is fairly straight forward and portable, while the other is to pin some of them to a certain hardware registers. This gives a considerable performance increase but at quite a large cost to the complexity of the back-end code generation. Also, the technique isn't portable, require a new virtual register to hardware register mapping for each architecture.

The RTS is written with these two modes in mind though, and in the case of registered code, it uses in-line assembly at times to operate on specific virtual registers by directly accessing the hardware register they are expected to be fixed to. This means that in the case of registered mode, the back-end code generators aren't easily able to vary from the current implementation of registered mode. Changing the virtual register mapping for example would require not just changing the back-end code generators but also some fairly tricky parts of the run-time system. This represents a considerable problem for in the implementation of an LLVM back-end, which will be discussed in *section 3.3.4*.

While the design of STG greatly effects the code that the back-ends must generate, it is of little direct concern for the implementation of a back-end code generator due to the presence of a final intermediate representation, *Cmm*, that STG is compiled to. The main influence of STG

[2] Please refer to `compiler/cmm/CmmExpr.hs` in the GHC source tree

STG Registers	
Sp	Stack pointer, points to the last occupied stack location
SpLim	Current stack size limit
Hp	Heap pointer, points to last occupied heap location
HpLim	Current Heap size limit
CurrentTSO	Pointer to current thread object
CurrentNursery	Pointer to the current allocation area
HpAlloc	Allocation count to check for heap failure
Argument and Return registers	
VanillaReg	Pointers, Ints and Chars
FloatReg	Single precision floating-point registers
DoubleReg	Double precision floating-point registers
LongReg	Long Int registers
Common Function Pointers	
EagerBlackholeInfo GCEnter1 GCFun	The address of some commonly-called functions are kept in the register table to keep code size down
Other	
BaseReg	Stores the address of the heap allocated STG registers. As not all STG virtual registers are pinned to hardware registers we need to still store the rest in memory.
PicBaseReg	Base Register for position-independent code calculations. Only used in the native code generator.

Table 2.1: STG Virtual Registers

is its effect on the design of *Cmm*, which primarily is the inclusion of the aforementioned STG virtual registers. We will now look at *Cmm*, the IR that the back-end code generators receive.

2.4.3 Cmm

The final intermediate representation used by GHC is the *Cmm* language. It serves as a common starting point for the back-end code generators. GHC's Cmm language is based on the C– language [21] but with numerous additions and removals from the language, the most import of these being that Cmm doesn't support any of C–'s run-time interface. More portable and direct techniques are instead used by GHC for implementing garbage collection and exception handling. They are implemented in such that no special support is required from the back-end code generators.

Cmm can be thought of in a similar fashion to the Low Level Virtual Machines language, both are a variation of a portable assembly language, with Cmm basing itself on C's syntax. Cmm abstracts away the underlying hardware and provides the following features:

- Unlimited variables, abstracting real hardware registers. The register allocator will either assign them to variables or spill them to the stack.

- Simple type system of either bit types or float types.
- Functions and function calling with efficient tail call support.
- Explicit control flow with functions being comprised of blocks and branch statements.
- Direct memory access.
- Powerful label type which can be used to implement higher level data types such as arrays and structures.

Cmm aims to represent Haskell in a low level procedural form as possible while still being abstracted from the underlying hardware that the Haskell program is being compiled for. It is the basis for all back-end work in GHC, with the C back-end and native code generator both translating from Cmm. It is from this point also that I implemented the LLVM back-end. *Cmm* greatly simplifies the task of a back-end code generator as the non-strict, functional aspects of Haskell have already been handled and the code generators instead only need to deal with a fairly simple procedural language.

An example of Cmm code can be seen below in *listing 2.1*:

```
1  fib()
2  {
3     bits32 count;
4     bits32 n2;
5     bits32 n1;
6     bits32 n;
7
8     n2 = 0;
9     n1 = 1;
10    n = 0;
11    count = R1;
12
13    if (count == 1) {
14          n = 1;
15          goto end;
16    }
17
18 for:
19    if (count > 1) {
20       count = count - 1;
21       n = n2 + n1;
22       n2 = n1;
```

```
23      n1 = n;
24      goto for;
25    }
26
27  end:
28    R1 = n;
29    jump StgReturn;
30  }
```

Listing 2.1: Cmm Example: Fibonacci Calculation (non recursive)

In this listing we can see a variety of *Cmm* features, such as:

- Use of *'bitsN'* notation to donate a bit type of a particular length.
- Use of branching and code blocks to implement high level constructs such as looping.
- Use of STG virtual register ***R1*** for receiving arguments and returning values.
- Use of a tail call. The final statement uses a *jump*, which denotes a tail call to a function.

One important feature of *Cmm* is its use of labels. These are a very powerful feature, behaving in a similar fashion to label in an assembly language. They mark a particular address in the code and are otherwise type free and unbound to any particular data. An example of how they are used can be seen below in *listing 2.2*:

```
1  section "data" {
2      sf5_closure:
3          const sf5_info;
4          const 0;
5      sf5_closure_mid:
6          const 0;
7          const 0;
8  }
9
10 section "readonly" {
11     cfa_str:
12         I8[] [72,101,108,108,111,32,87,111,114,108,100]
13 }
```

Listing 2.2: Cmm Example: Use of labels

In this listing, the first block of code defines what can be thought of as a structure type. However it is important to note that the two code labels don't act the same as variable names. They don't have any type information stored with them, all they do is store the address at the location they are defined. They are all immutable constants of the native data-pointer type and cannot be assigned to [21]. Because of this it is possible to put them at any point in a data structure, as in the above example where *'sf5_closure_mid'* can be used to access the middle of the structure. The second block in *listing 2.2* shows a string in *Cmm*, for which special language support is included, allowing them to be defined as arrays. Arrays are otherwise largely unsupported and must be created using the techniques demonstrated in the first block of code.

Relevant parts of the *Cmm* syntax will be introduced in *section 3* as required. It is important to keep in mind that while an overview of the rest of GHC is useful, it is not strictly required for the implementation of a new back-end code generator in GHC. Such is the design of *Cmm* that it allows the rest of GHC to be mostly ignored and a smaller area focused on, translating *Cmm* to the back-end target language.

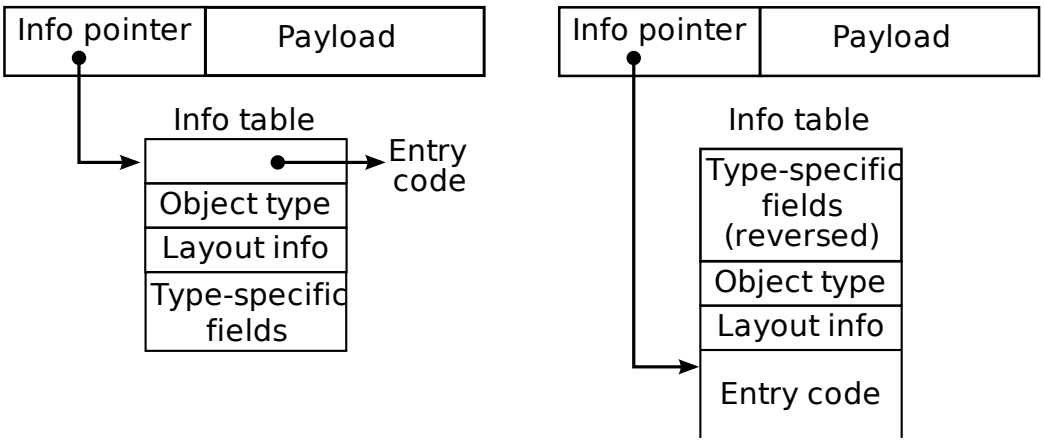
2.5 Layout of Heap Objects

In this section we will briefly look at how GHC lays out objects in the Heap. As functions are first class in Haskell they too stored on the heap as a closure objects and it is these that we are most interested in.

As GHC implements Haskell, all objects on the Heap share the same basic layout, that of a closure object. Some are statically allocated and other are dynamically allocated but all share the same layout as shown below in *figure 2.3a* [26].

As you can see in *figure 2.3*, there are two variants of heap layout that GHC can use, we will initially look at the *standard layout*. The first word is the object's info pointer, a pointer to an immutable, statically-allocated *info table*. The remainder of the object is the *payload*. Its contents are dependent on the closure type, in the case of a function or constructor object it contains the applied arguments for the function or constructor call. The info table itself contains an object-type field, used for distinguishing object kinds, and layout information for garbage collection purposes. Finally the info table also contains pointer to some executable code for the object, its purpose varies by closure type but in the case of a function object has the code for the function body.

GHC also supports an optimised version of the heap layout, which is shown in *figure 2.3b*. In this layout, the code for a object is placed adjacent to the info table, immediately after it. This allows for the info table pointer to be used for accessing both the info table and the



(a) Standard Layout (b) Optimised Layout

Figure 2.3: GHC Heap Layouts

objects entry code. This reduces the size of the info table by one word and saves one indirect jump when accessing an objects code. It does however place the requirement on back-end code generators of being able to control the layout of the final native code generated. In nearly all cases other then a native code generator this is very difficult to impossible. The GHC C back-end for example is unable to enforce this layout and requires the assembly produced by *gcc* to be post-processed to enable the optimised heap layout.

2.6 Low Level Virtual Machine

The Low Level Virtual Machine (*LLVM*) is a open source, mature optimising compiler framework that was started by *Chris Arthur Lattner* in 2000 as part of his Master thesis [24]. It provides a high performance static compiler back-end but can also be used to build Just-in-time compilers and to provide mid-level analyses and optimisation in a compiler pipeline. It currently supports over 9 different architectures and is under heavy ongoing development, with Apple being the primary sponsor of the project. Apple is continuing to make use of LLVM more and more, now as of *OS X 10.6* using it in their OpenGL and OpenCL pipeline and to compile certain performance critical parts of the operating system [17]. I have used it in this thesis to implement a new back-end code generator for GHC.

LLVM offers a good compiler target and platform for a Haskell code generator for a number of reasons, in particular:

- LLVM is distributed under the Illinois Open Source License, which is nearly identical to the BSD license. This is a very open licence which allows it to be freely used by both open source and proprietary projects. GHC itself is distributed under a nearly identical licence.

- LLVM provides a high performance code generator for a variety of platforms, including all of those currently supported by GHC.
- LLVM has a very active development community and is quickly progressing in all areas.
- LLVM provides a very well designed platform for back-end code generation and optimisation work. It also provides a lot of interesting technologies for GHC developers to play around with, such as an interpreter with just-in-time compilation and static analysis tools. The LLVM code base itself is very well written and quite easy to change.
- LLVM has native vector support. While this isn't currently used in the LLVM back-end its of considerable interest to GHC developers.

Its for these reasons and the deficiencies outlined in *section 2.3.1 and 2.3.2* that I have implemented a new LLVM back-end for GHC. Let us now look though at the assembly language provided by LLVM so as to get an understanding of the task required to compile from Cmm to LLVM Assembly.

2.6.1 LLVM Assembly Language

LLVM's assembly language is the input language which LLVM accepts for code generation. However it also acts as LLVM's internal intermediate representation (*IR*) when performing optimisation passes and other analysis. The IR has three equivalent forms: text (the assembly form), in-memory, and binary. It is designed around the idea of being a low level representation, a portable assembly, but supporting high level type information. The reason for this is to provide an ideal compromise between retaining enough information in the code so as to be able to aggressively optimise, while still being low level enough to efficiently support a wide variety of programming languages [24].

The features of LLVM's assembly language are:

- Low level instruction set with high level, strong types
- Single static assignment form (*SSA*) with Phi (Φ) function
- Infinite amount of virtual registers, abstracting real hardware registers
- Unified memory model through explicit heap and stack allocation
- Function support with efficient tail calls

SSA form means that once a value has been assigned to a virtual register the virtual register becomes immutable, permanently storing the value it was initially assigned. The phi (Φ) function is used to implement control flow in a SSA form language by selecting one value from a list to assign to a virtual register, based on the where in the control flow graph the flow came from. This can be seen in *listing 2.4* which is an equivalent LLVM program of the C code in *listing 2.3*. The reason for the use of SSA form is to facilitate optimisations by not allowing aliases to occur.

```
1 int pow(int M, unsigned N) {
2         unsigned i;
3         int Result = 1;
4
5         for (i = 0; i != N; ++i)
6                 Result *= M;
7
8         return Result;
9 }
```

Listing 2.3: C Example: Raise to power

The LLVM code which follows in *listing 2.4* succinctly represent the core of LLVM's IR. The listing shows one complete function in LLVM, which is made up of a list of basic blocks, each one being denoted by a label. The function has three basic blocks, those being `LoopHeader`, `Loop` and `Exit`. All control flow in LLVM is explicit, so each basic block must end with a branch (`br`) or return statement (`ret`). For instructions, the left hand side of assignment statements store to *virtual registers*, so `%res, %res2, %i, %i2` and `%cond` are all virtual registers. Virtual registers can be introduced as needed by assigning to them, no other declaration is needed. Notice how due to the SSA form, new registers must be used for all operations as each virtual register can only be assigned to once. The high level type information of LLVM can also be seen in every operation, for example on line 6 the `i32` keyword sets the virtual `%res` to be an integer type of length 32. Finally, we can also see the Phi (Φ) function in use on line 6. This sets `%res` to be either the value of `1` or the value stored in register `%res2` depending on if we entered this block (`Loop`) from the block `LoopHeader` or from the block `Loop` (that is from itself). As stated before, the Phi (Φ) function selects from a list of values depending on where the control flow came from.

```
1 define i32 @pow(i32 %M, i32 %N) {
2         LoopHeader:
3                 br label %Loop
4
```

```
5         Loop:
6               %res  = phi i32 [1, %LoopHeader], [%res2, %Loop]
7               %i    = phi i32 [0, %LoopHeader ], [%i2, %Loop]
8               %res2 = mul i32 %res, %M
9               %i2   = add i32 %i, 1
10              %cond = icmp ne i32 %i2, %N
11              br i1 %cond, label %Loop, label %Exit
12
13        Exit:
14              ret i32 %res2
15  }
```

Listing 2.4: LLVM equivalent of *listing 2.3*

All LLVM code is defined as part of a module, with modules behaving in a similar fashion to object files. An LLVM module consists of four parts, meta information, external declarations, global variables and function definitions. Meta information can be used to define the endianness of the module, as well as the alignment and size of various LLVM types for the architecture the code will be compiled to by LLVM. Global variables function as one would expect, and are prefixed with the `@` symbol, as are functions, to indicate that they are actually a pointer to the data and have global scope. This also distinguishes them from local variables which are prefixed with the `%` symbol, as can be see in *listing 2.4*. Global variables can be marked as *constant* and provided with data, making them immutable. All global variables and functions also have a linkage type, which determines how they will be handled by the system linker when compiled to a native object file. As mentioned before, functions themselves are made up of a list of instruction blocks which explicitly define the control flow graph of a function. The blocks themselves are each made up of a list of sequentially executed instructions, ending with a branch or return statement.

2.6.2 LLVM Type System

A unique feature of LLVM is its high level type system, designed to give enough information to the LLVM optimiser and compiler so that they can produce optimal code. Below in *table 2.2* is a brief listing of LLVM's type system [3] [31].

LLVM requires that all instructions be decorated with type information, not inferring anything from the types involved in the instruction. It also requires all type conversion be explicitly handled, providing instructions for this purpose even though many of them will be compiled to `no-op` instructions.

[3] This is not the full LLVM type system. A few types (meta-data, packed structures, vectors and opaque types) are missing as they aren't relevant to the work done in this thesis.

Type	Syntax	Description
Primitive Types		
Integer	`i1`, `i2`, ... `i32`, ...	Simple bit type of arbitrary width.
Floating Point	`float`, `double`, ...	Usual array of floating point types.
Void	`void`	Represent no value and has no size.
Label	`label`	Represents code labels for control flow.
Derived Types		
Array	`[40 x i32]`	A simple derived type that arranges elements sequentially in memory. As a special case zero-length arrays are allowed and treated as variable length.
Structure	i32, i32	A collection of data stored in memory together.
Pointer	<type> *	The pointer type represents a reference or the address of another object in memory. Pointers to void or labels are not allowed.
Function	i32 (i8*, ...)	A function type, consisting of a return type and formal parameter types. The function in the example takes at least one pointer to an i8 type and returns an integer.

Table 2.2: LLVM Type System

2.6.3 LLVM Instruction Set

In this section we give a brief overview of a selection of the available LLVM instructions. The purpose is to give enough information here to allow LLVM code to be read and understood, and to be able to understand the design of the LLVM back-end code generator, detailed in *section 3*. The instructions are presented in related groups as they are in the LLVM assembly language reference [31].

Terminator Instructions

Terminator instructions deal with control flow. As mentioned previously, all basic blocks in LLVM must end with a terminator instruction.

- `ret`: Return control flow from the function back to the caller, possibly returning a value.
- `br`: Used to transfer control to another basic block. Can be conditional or unconditional.
- `switch`: Used to transfer control to one of several possible basic blocks. Will usually either be compiled to a jump table or a serries of conditional branches.

Binary Operations

These operates are used to do most the computation in LLVM. They require two operands of the same type and compute a single value from them, also of the same type as the operands. As well as operating on scalar values, they can also operate on vectors. Below is a list of the operations.

- `add`, `sub`, `mul`: Perform addition, subtraction and multiplication respectively. Overloaded to handle integer, floating point and vector types.
- `udiv`, `sdiv`, `fdiv`: Perform division on unsigned integers, signed integers and floating point types respectively. Can also handle vector types.
- `urem`, `srem`, `frem`: Return the remainder from a division operation for unsigned integers, signed integers and floating point types respectively. Can also handle vector types.

Bitwise Binary Operations

Bitwise operators are used to perform bit manipulation. As with binary operations, they expect two operands of the same type and return a result of the same type. They are only applicable to integer or integer vector types though.

- `shl`, `lshr`, `ashr`: Perform a bit shift operation, left-shift, logical right-shift, and arithmetic right-shift respectively.
- `and`, `or`, `xor`: Perform logical and, or and xor bitwise operations respectively.

Memory Access and Addressing Operations

In LLVM, no memory locations are in SSA form but are instead mutable. However, all memory is accessed through pointers which are themselves in SSA form. LLVM includes strong support for memory accessing and addressing with all memory operations being explicit.

- `alloca`: Allocates memory on the stack frame of the current function. Is automatically released when the function exits by returning to its caller or through a tail call.
- `load`: Used to read from memory, a pointer and type for the data to load must be given.
- `store`: Used to write to memory, a pointer and value to write must be given.
- `getelementptr`: This instruction is used to get the address of a sub-element of an aggregate data structure such as an array. It is important to note that this instruction only performs address calculations. It never actually touches memory.

Conversion Operations

Given LLVM's high level type system and strict enforcement of it, a number of conversion operators are needed to convert between types. All type conversion must be done explicitly by using these operators, even when they will be removed during compilation as no bit changing is required.

Other Operations

- `icmp, fcmp`: Perform a variety of integer and floating point comparisons respectively. Return an `i1` or Boolean type which can be used in a `br` instruction. Instructions include a comparison condition and two operands.
- `phi`: As previously explained the `phi` instruction selects a value from a list according to which predecessor block the control flow came from.
- `call`: The `call` instruction is used to perform function calls. It can take an optional `tail` marker to indicate the call should be tail call optimised. It also supports declaring the call convention which should be used.

2.7 Cmm and LLVM Compared

In this section I will give a brief outline of the similarities and differences between the Cmm language and the LLVM Assembly language. They are for the most part remarkably similar, which isn't of that great surprise when one considers that both had the same intentions behind their design.

Cmm as outlined previously is based on the *C–* language, which is language designed with the same goal LLVM of providing a portable, high quality code generation back-end for compiler writers to target. While there are several important differences in the approach of C– and LLVM, these are largely external from the language design. C– supports a sophisticated API and run-time-system to allow for high level services to be implemented in an efficient and portable manner. LLVM provides an aggressive optimisation framework, particular focusing on advance link-time optimisation techniques. In both cases though this doesn't affect the features provided by the core language too greatly, as both still aim to provide a low level portable assembly target.

Both provide nearly identical sets of primitive operations, and both use nearly the same control flow constructs, using functions comprised of basic blocks and supporting only branch and switch instructions. Looping in both cases must be done with branches to labels. Both provide similar function declaration and calling support, including efficient tail calls. Both provide unlimited local variables to abstract over hardware registers.

One area of interest though is the memory addressing capabilities of both. Syntactically they are quite different, with Cmm using labels exclusively for addressing similar to an assembly language and LLVM using variable names and the `getelementptr` instruction. However, semantically both are very similar, both resulting in immutable memory addresses that must be calculable at compile time. There is one other difference though, *Cmm* has no concept of an external declaration. Instead it simply allows external labels to be referred to and leaves it up to the

system linker to resolve them. LLVM requires that all types be defined, so external references must be declared and with a type.

There are two main differences between the languages, Cmm's inclusion of the STG virtual registers (outlined in *section 2.4.2*) and Cmm's support for data layout. Cmm allows for some control over how the final generated code should be laid out. This is used as explained in *section 2.3* to implement the *TABLES_NEXT_TO_CODE* optimisation. Both of these features are used in GHC's *registered* mode and cause considerable problems for the LLVM back-end.

2.8 Related Work

Currently there is a lot of working taking place in the compiler community with regards to the Low Level Virtual Machine (*LLVM*). There are numerous project in various stage of completion using LLVM either as a static back-end compiler as I have in this thesis, or as a just-in-time compiler for an interpreted language running on a virtual machine. There are also a few projects using LLVM for static analysis, performing no compilation work at all. From all these uses we can gain an idea of the powerful and versatile system LLVM provides and the possibilities it brings to a research community like GHC.

2.8.1 Essential Haskell Compiler

The Essential Haskell Compiler (*EHC*) is a research Haskell compiler, designed around the idea of implementing the compiler as whole serries of compilers [12]. It is an experimental Haskell compiler largely designed by members of the Department of Information and Computing Sciences at Utrecht University.

An experimental LLVM back-end for EHC was recently implemented by *Joh Van Schie* as part of his thesis for a Master of Science, with the results being published in the paper, *Compiling Haskell To LLVM* [32]. The comparison with GHC is particularly relevant as EHC's usual back-end is a C code generator, very similar to GHC's C back-end. His work produced some impressive results, which included:

- A 10% reduction in compilation times
- A 35% reduction in generated code size
- A 10% reduction in the runtime of the code.

The LLVM back-end however didn't reach the stage of being able to handle the full Haskell language, instead working with a reduced subset of Haskell.

To put these results in to perspective it is useful to compare EHC with GHC. Below in *table 2.3* are the results from the *Compiling Haskell To LLVM* paper used to evaluate the EHC LLVM back-end and produce the above mentioned results. It has been updated though to include the results of running these tests against GHC and also against EHC on my test machine, attempting to replicate the results in the paper as best as possible. The benchmark is based on the nofib [28] benchmark suite but a reduced subset which can be handled by EHC. The original benchmarking from the *Compiling Haskell To LLVM* paper were performed on a machine with an Intel Core2 processor and 3.2 gigabytes of memory, running 64 bit Linux with a 2.6.24 kernel. The benchmarking of EHC against GHC was done on a very similar machine, an Intel Core2 processor running at 2.4 Ghz and 3.4 gigabytes of memory, running 32 bit Linux with a 2.6.28 kernel.

	Original EHC Results			EHC Vs. GHC[1]		
	EHC C	EHC LLVM	ratio	EHC C	GHC	ratio
digits-of-e1	8.50	7.27	1.17	8.41	0.09	93.40
digits-of-e2	5.26	6.53	0.81	5.52	0.08	69.00
exp3_8	0.47	0.50	0.93	0.48	0.03	16.00
primes	0.89	0.88	1.02	0.84	0.06	14.00
queens	0.93	0.75	1.24	1.43	0.10	14.30
tak	0.22	0.15	1.45	0.24	0.02	12.00
wheel-sieve1	7.85	7.18	1.09	7.81	0.08	97.63
wheel-sieve2	0.65	0.62	1.05	0.66	0.09	7.33
average			1.10			40.46

Table 2.3: Execution time (sec) of select programs from nofib benchmark suite

As we can see from *table 2.3*, GHC generates far more efficient code then EHC currently does. This gave the EHC LLVM back-end far more room to work with to produce more efficient code then the EHC C back-end. GHC's current code generators on the other hand generate code approaching near optimal, giving the LLVM back-end a hard task of achieving these kinds of results.

2.8.2 GHC Related Work

In addition to the new LLVM back-end implemented as part of this thesis, there is also some other related work going on within the GHC community. In particular there is work being done

[1] GHC was built in registered mode and run using the native assembly generator back-end with an optimisation level of O2.

by Norman Ramsey, Simon Marlow, Simon Peyton Jones, and John Dias to re-architecture GHC's back-end pipeline [29]. This represents a considerable amount of work which has been done and is still ongoing. This work is currently been done outside of the main GHC development.

A large part of the future work planned is the design and implementation of a new native code generator that implements a variety of optimisation passes such as constant propagation, partial redundancy elimination and dead code removal. I believe that the new LLVM back-end represents a better choice then implementing a new native code generator as it is ready for use today and already provides many of the optimisations that the new back-end hopes to be able to perform, including all those aforementioned. This work was managed by myself alone in a reasonably short time frame, while the work by Ramsey et al. has been going for over a year now. The LLVM back-end will also benefit from the ongoing efforts of the LLVM developers, improving the code generated by GHC for free and allowing the GHC developers to focus on more interesting problems.

2.8.3 Other Projects

There are a large amount of related projects also using LLVM in various capacities, from static compiler and interpreters to static analysis tools. These help to give an indication of the power, flexibility and richness of the LLVM compiler infrastructure. It also shows the size and health of the LLVM community. A short list of related projects which are using LLVM include:

- *Clang*: A C, C++ and Objective-C compiler using LLVM as a back-end target [1]. The clang project is using LLVM to produce a static analyser for C and Objective-C programs which can be used to automatically find bugs.
- *OpenJDK Project Zero*: A version of Sun Microsystems open source JVM, *OpenJDK*, which uses zero assembly. LLVM is used as a replacement for the usual just-in-time compiler [5].
- *Pure*: Pure is an algebraic/functional programming language based on term rewriting. It uses LLVM as a just-in-time compiler for its interpreter. [15]
- *MacRuby*: A Ruby implementation with an LLVM based just-in-time compiler [4].
- *Unladen Swallow*: Google backed Python virtual machine with an LLVM based just-in-time compiler [7].
- *LDC*: A compiler for the D programming language using LLVM as a back-end for code generation. [2]

- *llvm-lua*: A compiler for the Lua programming language using LLVM as both a just-in-time compiler and for static compilation. [3]

3

The LLVM Back-end

A large part of the work of this thesis was the design and implementation of a new back-end for GHC which outputs LLVM assembly. In this chapter we discuss the approach taken to achieve this.

3.1 Pipeline Design

The overall approach is to add in a new LLVM backend at the end of the GHC pipeline, placing it at the same place where the two current GHC back-ends are. This can be seen previously in *figure 2.2*. More important than this though is the design of the GHC pipeline from this stage on. The design of the new LLVM back-end pipeline and the existing C and Native Code Generator (*NCG*) pipelines can be seen below in *figure 3.1*.

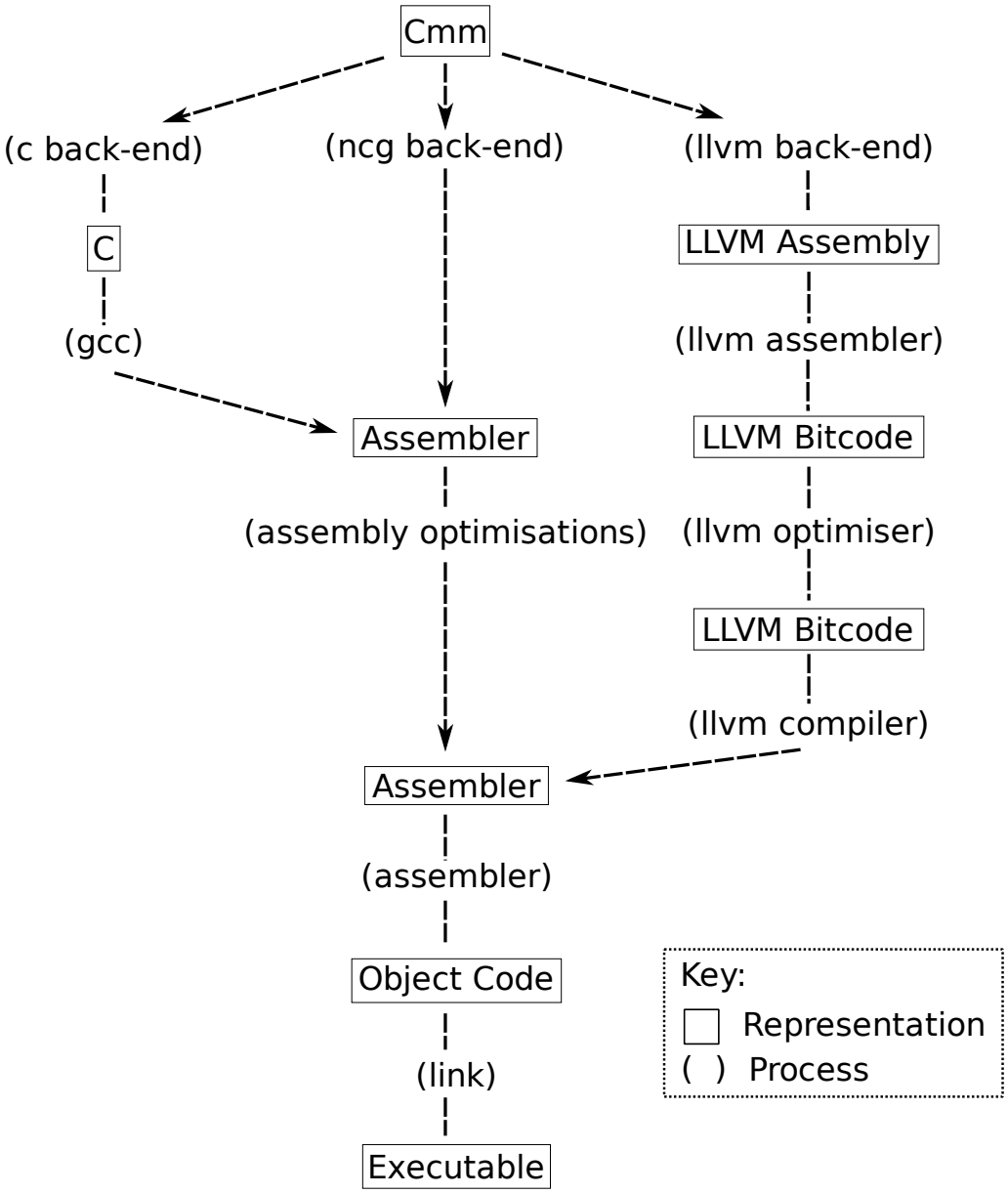


Figure 3.1: Compiler Back-end Pipeline

The primary rational behind the design of the new LLVM pipeline was for it to fit in with GHC's existing back-end structure as seamlessly as possible. Allowing for quicker development of the pipeline and for focusing on the core task of LLVM code generation, which is required regardless of the design of the back-end pipeline.

As we can see from *figure 3.1* both the C back-end and NCG produce native assembly for the machine the code will run on (the target architecture), which is then optimised before being passed to the system assembler to produce a native object file. At the end of the pipeline once all code has been compiled to native object files, the system linker is invoked to combine them all and link in the Haskell run-time-system. As we can see a large part of this pipeline is shared by the C and NCG.

The LLVM back-end attempts to rejoin this shared pipeline as soon as possible, reducing the amount of work required in its implementation. This point occurs right after assembly level optimisations have occurred, when the representation is native assembly code. The LLVM pipeline can't rejoin the main code generation path earlier then this as the assembly optimisation phase is not a general assembly optimiser but specific to the assembly code produced by GCC and the NCG. We will now quickly look at the steps involved in the LLVM pipeline, although these will be explained in more detail later in the chapter. Data is passed from stage to stage using on-disk temporary files.

- ***llvm back-end***: This is where the bulk of the work is done, translating from *Cmm* to *LLVM Assembly*. This is examined in *section 3.3*.
- ***llvm assembler***: In this stage the *LLVM Assembly* language is transformed into *LLVM Bitcode*. An equivalent binary form of the assembly language that the rest of the LLVM infrastructure works with. This is examined in *section 3.4.1*.
- ***llvm optimiser***: Here LLVM's impressive optimisations are applied to the *Bitcode*. This stage is optional and only used if specified by the user. This is examined in *section 3.4.2*.
- ***llvm compiler***: In the final stage, the LLVM compiler is invoked, producing native assembly for the *Bitcode* which can then be processed by the standard GHC pipeline. This is examined in *section 3.4.3*.

Before any of this though we must look at how LLVM Assembly Code is generated from Haskell as part of the LLVM back-end.

3.2 LLVM Haskell Representation

A requirement of generating LLVM code is to have a method for interfacing with the LLVM optimisers and code generators, this is what the LLVM Assembly language is used for. But how should this be represented in Haskell so that it can be generated? The *LLVM FAQ* suggest three possible approaches [22]:

- Call into LLVM Libraries using Haskell's Foreign Function Interface (*FFI*).
- Emit LLVM assembly from Haskell.
- Emit LLVM bitcode from Haskell.

For the first approach there already exists an binding for Haskell to the LLVM API [27]. This approach was decided against though as the existing binding was quite large and operated at a very high level, attempting to bring Haskell's powerful type safety to the LLVM API. While this may work well when using them in a new project, they were too big and bulky for fitting in well with the existing GHC code base.

The third approach was also rejected as it involved a fair amount of complex and boring work to produce the required binary output, with the right bits and alignments. There is also very little reason to take this approach, the only one being that it offers ever so slightly faster compilation times over the second approach as the *LLVM assembler* phase can be removed from the LLVM back-end pipeline.

In the end I went with the second suggested approach, emitting LLVM assembly from Haskell. This is the same approach taken by the *Essential Haskell Compiler* and was a major motivation for my decision as I was able to use the LLVM assembly library from EHC as part of the LLVM back-end for GHC. This library contains a first order abstract syntax representation of LLVM Assembly and the ability to pretty print it. I have heavily modified it to increase its language coverage as it was missing several LLVM constructs which were needed in the new GHC back-end. I also went with this approach as it allows for a complete LLVM module to be easily represented and analysed in memory in Haskell. As part of future work I would like to modify this library so that through Haskell's FFI, it uses the LLVM libraries to generate LLVM Bitcode instead of LLVM Assembly.

3.3 LLVM Code Generation

In this section we explore the generation of equivalent *LLVM Assembly Code* from *Cmm*, the final intermediate representation used by GHC to represent Haskell code. It is assumed that the reader is somewhat familiar with the Haskell programming language and is able to understand Haskell code. The first issue that must be dealt with is the difference between *registered* and *unregistered* Cmm code. We looked at the difference between the two from a high level perspective in *section 2.3* but need to now look at the difference from a low level code generation viewpoint (*section 3.3.1*). Then we will look at the code generation process. This is broken into two major tasks, handling *Cmm global data* (*section 3.3.2*) and handling *Cmm functions* (*section 3.3.3*). As *unregistered* code is a subset of *registered* code, these two sections will detail the handling of only *unregistered* code. The extra work required to handle *registered* code is the subject of *section 3.3.4*.

First though we must look at the code generation work which occurs before these two tasks. Code generation consists of translating a list of `RawCmmTop` data types to LLVM code. `RawCmmTop` is a Haskell data type which represents a top level Cmm construct. It has the following form:

```
data GenCmmTop d h g
  = CmmProc           -- Function
        h             -- Extra header such as the info table
        CLabel        -- Procedure name
        CmmFormals    -- Argument locals live on entry
        g             -- Control flow graph

  | CmmData           -- Static data
        Section       -- Type
        [d]           -- Data

data GenBasicBlock i = BasicBlock BlockId [i]

newtype ListGraph i = ListGraph [GenBasicBlock i]

type RawCmmTop = GenCmmmTop CmmStatic [CmmStatic] (ListGraph CmmStmt)
```

Listing 3.1: Cmm Top Definition

It consists of two types, static data and functions, each of which can largely be handled separately. Just enough information is needed such that pointers can be constructed between them, for this purpose top level labels need to be tracked. *CmmProc* and *CmmData* will each be examined in more detail in *section 3.3.2* and *section 3.3.3* respectively.

A large part of the code generation work is keeping track of defined static data and functions. This is needed for the LLVM back-end as all pointers must be created with the correct type, `void` pointers are not allowed. To create a pointer to a variable in LLVM, the `getelmentptr` instruction is used and it requires that the type of the variable be specified as part of the instruction. An example usage of it can be seen below in *listing 3.2*.

```
@x = global [4 x i32] [ i32 1, i32 2, i32 3, i32 4 ]

define i32 @main() {
entry:
        ; retrieve second element of array x
       %tmp1 = getelementptr [4 x i32]* @x, i32 0, i32 2
        ...
```

Listing 3.2: Usage of LLVM `getelementptr` instruction

To keep track of the defined global variables and functions a fairly simple environment is used. The environment simply stores a mapping between defined variable and function names and

their types. It consists of two such mappings though, one for all global variables and functions and one that is local to a particular function and keeps track of local variables. The function of the second map is explained more in *section 3.3.3*. Below in *listing 3.3* is the Haskell code used to implement the environment described here. In this code, `Map.Map` is simply a associative array and `LMString` is just a String type.

```
1 type LlvmEnvMap = Map.Map LMString LlvmType
2
3 -- two maps, one for functions and one for local vars.
4 type LlvmEnv = (LlvmEnvMap, LlvmEnvMap)
```

Listing 3.3: LLVM Back-end Environment

A single environment is used for code generation, although the local variable mapping is replaced for each new function generated. It is currently passed around in the back-end code as a function argument but should really be implemented as a Monad.

3.3.1 Registered Vs. Unregistered Code

Code generation can take place in two general modes, unregistered and registered. Registered mode is a superset of unregistered in terms of the requirements placed on a back-end code generator for GHC. In registered mode a optimisation features called *TABLES_NEXT_TO_CODE* is enabled and the STG virtual registers are pinned to physical hardware registers. Both of these optimisations are outlined in *section 2.3*.

Firstly, when the *TABLES_NEXT_TO_CODE* optimisation is enabled in a registered build, this means that the first field of the `CmmProc` constructor, the algebraic `h` type, is populated with a list of `CmmStatic` types that make up the functions so called 'info table'. These values must be compiled to code, which by itself isn't difficult, but the back-end code generator must also guarantee that they are placed just before the procedure in the native code. In unregistered mode the `h` field is instead populated with an empty list and the 'info table' for the function is compiled as a `CmmData` object.

The other major change is the use of pinned global registers. The defection and use of these registers is explained in *section 2.4.2*. Cmm includes two types of registers which are defined in GHC as shown below in *listing 3.4* [1] [11].

```
1 data CmmReg
```

[1] Please refer to `compiler/cmm/CmmExpr.hs` in the GHC source tree

```
2     = CmmLocal    LocalReg
3     | CmmGlobal  GlobalReg
4     deriving( Eq, Ord )
```

Listing 3.4: GHC Code: CmmReg data type

A `LocalReg` is a temporary general purpose registered used in a procedure with scope of a single procedure. A `GlobalReg` however has global scope and corresponds to one of the STG virtual registers outlined in *table 3.1*. In *unregistered* mode they are replaced by load and stores to memory. However in registered mode they are retained and read and writes to them must be compiled to reads and writes to the register they are mapped to. The back-end must also ensure that the STG virtual registers are not clobbered and can always be found in the expected hardware register when needed. Both the C and NCG back-end do this by exclusively reserving the needed hardware registers for use by the mapped STG virtual register. The mapping between STG registers and hardware registers is unique for each architecture, *table 3.1* below though shows that mapping for *x86*.

STG Register	x86 Register
Base	`%ebx`
Sp	`%ebp`
Hp	`%edi`
R1	`%esi`

Table 3.1: x86 Mapping of STG Virtual Registers

To give a clear understanding of the differences between *unregistered* and *registered* code, *listing 3.5* and *listing 3.6* show the equivalent *Cmm* code but in *unregistered* and *registered* form respectively.

```
1  section "data" {
2      :Main.main_info:
3          const :Main.main_entry;
4          const 0;
5          const 196630;
6          const :Main.main_srt;
7  }
8
9  :Main.main_entry()
10         { []
11         }
12     cmv:
13         if (I32[MainCapability+92] - 12 < I32[MainCapability+96]) goto cmx;
14         I32[MainCapability+100] = I32[MainCapability+100] + 8;
```

```
15                                    ...
```

Listing 3.5: Cmm Unregistered Code

In *listing 3.5* above, `MainCapability` is a label to the start of a run-time-system defined structure that stores all the global registers. This structure resides in memory on the heap. The offsets from it represent accessing particular STG virtual registers.

```
1  :Main.main_entry()
2          { [const :Main.main_srt-:Main.main_info;, const 0;, const 196630;]
3          }
4      cg6:
5          if ((Sp + -12) < I32[BaseReg + 84]) goto cg8;
6          Hp = Hp + 8;
7                                    ...
```

Listing 3.6: Cmm Registered Code

3.3.2 Handling CmmData

CmmData is one of the two top level objects for *Cmm*, it specifies static data which should be generated. A *CmmData* object has two fields, a single value of type *Section* and a list of type *CmmStatic*. This is shown below in *listing 3.7* [2] [11].

```
1  -- CmmData = CmmData Section [CmmStatic]
2
3  data CmmStatic
4    = CmmStaticLit       CmmLit  -- a static value
5    | CmmUninitialised   Int     -- n bytes of uninitialised data
6    | CmmAlign           Int     -- align to next N byte boundary
7    | CmmDataLabel       CLabel  -- label current position in code (definition)
8    | CmmString          [Word8] -- string of 8 bit values
9
10 data CmmLit
11   = CmmInt             Integer  Width     -- 2 compliments, truncated int
12   | CmmFloat           Rational Width     -- float
13   | CmmLabel           CLabel             -- address of label (usage)
14   | CmmLabelOff        CLabel Int         -- address + offset
15   | CmmLabelDiffOff    CLabel CLabel Int  -- address1 - addres2 + offset
```

[2] Please refer to `compiler/cmm/Cmm.hs` in the GHC source tree

```
16    | CmmBlock          BlockId          -- address of code label
17    | CmmHighStackMark                   -- max stack space used during a proc
```

Listing 3.7: CmmData Type

The *Section* type specifies an object section that the data should be placed in, such as `readonly`. It is of little interest here, however the code generation for the *CmmStatic* list is. It occurs in two phases, firstly the types and all data is generated except for address values, that is any *CmmLabel...* types. Then the labels are resolved. This two step method is used as in the first pass, we don't know if the label is defined in this module or in an external module. As *CmmLabel's* can be declared at nearly any time, we must wait till we run one complete pass over all the data before we can begin resolving any address references. We also need the type information for all *CmmLabel's* in order to create a pointer in LLVM as outlined in *section 3.3*.

3.3.2.1 1st Pass : Generation

In this pass, all *CmmStatic* lists are translated to LLVM structures. We will now look at how each top level CmmStatic object type is handled.

CmmStaticLit

These are translated as follows:

- ***CmmInt*** → Reduced to Int and then an appropriate LLVM Int of correct size is created. As LLVM supports any bit size, this is very straight forward.
- ***CmmFloat*** → LLVM supports a number of different ways of encoding floating point types but the one used is the Hexadecimal encoding, the most flexible one. This format is basically a specification at the bit level of the floating point number as represented in the usual IEEE754 form [8]. As the code for this conversion already existed in the Native Code Generator this was the most straight-forward choice. The hexadecimal form is also needed to handle special floating point values such as *INFINITY*, so this was really the only choice. The other notary aspect of floating point conversion is that in LLVM floating point values are always in big-endian form. As the method of retrieving the hexadecimal form retrieves it in the machine format that GHC is running on, a compiler flag was needed to determine if the bytes needed to be reversed or not.
- ***CmmLabel*** → Left untranslated at first, later resolved once we have determined types. As pointers are cast to word size ints, we can still determine the type of a structure that contains a pointer, before the pointers type itself is resolved.
- ***CmmLabelOff*** → As above.

- ***CmmLabelDiffOff*** → As above.
- ***CmmBlock BlockId*** → is translated to an equivalent CmmLabel type and then handled as one.
- ***CmmHighStackMark*** → This type is not currently handled as it is not used by GHC in code generation. Both the C and NCG back-ends do the same.

CmmUninitialised

For this, a zeroed array of 8bit values is created of correct size.

CmmAlign & CmmDataLabel

The LLVM back-end can't handle CmmAlign or CmmDataLabel. A panic occurs if either is encountered. A CmmDataLabel is expected at the very start of each list of 'CmmStatic'. It is removed and used as the name for the generated structure. While it would be possible to handle CmmDataLabel correctly in all cases, not just the start of the list, this is not required as currently GHC doesn't generate any code which uses a CmmDataLabel at any other place. The C back-end would also panic in the same situation as outlined here.

CmmString

This is compiled to an LLVM array of 8 bit values. This approach was taken as it is the least error prone and the quickest route given its a direct mapping from the *Cmm* approach to Strings which are also arrays of 8 bit values.

3.3.2.2 2nd Pass : Resolution

After the first pass, all types have been determined and all data translated except for address values (*CmmLabel... types*). All generated LLVM data is added to the current environment and all *CmmProc's* are added to the environment as well, they don't need to be properly passed though, just their names retrieved as they have a single, fixed function type.

Now appropriate pointers can be generated using the type information from the environment and LLVM's `getelementptr` instruction. Once a pointer has been created it is immediately converted to an integer type. This is done to allow for all structure types to be determined in stage one as any pointers can be assumed to be simply a word size integer. If this cast wasn't done then determining the types of structures could not be done without resolving any pointers they contain, which turns the simple two pass approach into a complex recursive procedure.

If a pointer doesn't have a match in the environment then it is assumed to refer to an external (outside of this module) address. An external reference is declared for this address as as follows in *listing 3.8*.

```
@label = external global [0 * i32]
```

Listing 3.8: LLVM External Reference

Where `i32` is the native pointer size.

3.3.3 Handling CmmProc

In this section we will look at how *Cmm* procedures are compiled to LLVM code. A Cmm procedure is made up of a list of basic blocks, with each basic block being comprised of a list of *CmmStmt's*. While Cmm procedures include a specification for arguments and a return type there is in fact only one type used, that is a procedure which takes no arguments and returns void. The reason for this is that the STG registers are instead used for the purpose of argument passing and the returning of results. Another detail of the Cmm code produced by GHC is that it doesn't contain any return statements. Instead a style of code called continuation passing is used in which the control is explicitly passed in the form of a continuation, and all Cmm procedures produced by GHC are instead terminated by tail calls.

Below in *listing 3.9* [3] [11] is the Haskell definition for Cmm statements and expression [4]. This is the core of a Cmm procedure which must be compiled to LLVM assembly code. Rather then detail how each statement is compiled, I will give a brief overview of the process and then focus on the more interesting cases, as by and large the majority of Cmm maps quite straightforwardly onto LLVM.

```
data CmmStmt
  = CmmAssign ...       -- Assign to register
  | CmmStore ...        -- Assign to memory location.
  | CmmCall ...         -- A call
  | CmmBranch ...       -- Unconditional branch
  | CmmCondBranch ...   -- conditional branch
  | CmmSwitch ...       -- Table branch
  | CmmJump ...         -- Jump to another C-- function,

data CmmExpr,
  = CmmLit ...        -- Literal value
  | CmmLoad ...       -- Read from a memory location
  | CmmReg ...          -- Read contents of register
  | CmmMachOp ...     -- Machine operation (+, -, *, etc.)
```

[3] Please refer to `compiler/cmm/Cmm.hs` and `compiler/cmm/CmmExpr.hs` in the GHC source tree

[4] This is a simplified version with all fields and some irrelevant constructors removed

Listing 3.9: Cmm Statement and Expressions

A comparison of the *Cmm* statement and expressions shown here to the LLVM language detailed in *section 2.6.3* reveals the similarity of the two. As Cmm statements by and large include at least one expression, we will discuss how they are handled first and then statements after.

3.3.3.1 Cmm Expressions

CmmExpr's are handled in a relatively straight-forward manner. The most interesting aspect of their compilation to LLVM is the return type of functions in the LLVM back-end which compile *CmmExpr's*. This gives an idea of the compilation process, as while each expression must be handled differently, they all return the same type when compiled to LLVM code by the back-end. This shared return type is shown below in *listing 3.10*.

```
1 type ExprData = (LlvmEnv, LlvmVar, LlvmStatements, [LlvmCmmTop])
```

Listing 3.10: LLVM Compilation type for a Cmm Expression

- `LlvmEnv`: During code generation for an expression, an external *Cmm Label* may be encountered for the first time. An external reference for it will be created and return as part of the *[LlvmCmmTop]* list. It is also added to the current environment.
- `LlvmVar`: All expressions share the property that there execution results in a single value which can be stored in a variable. This LLVM local variable holds the result of the *CmmExpr*. This allows for statements to very easily use and access the result of an expression.
- `LlvmStatements`: A *CmmExpr* may require several LLVM statements to implement, they are returned in this list and must be executed before the *LlvmVar* is accessed.
- `[LlvmCmmTop]`: An externally declared *Cmm Label* can be encountered at any point as Cmm requires no external declaration. LLVM though requires that these labels do have an external declaration and in this list such declarations are returned. They add new global variables to the LLVM module of the form outlined in *listing 3.8*.

3.3.3.2 Cmm Statements

Statements are also handled in a fairly straight-forward manner process involved can be detailed most simply by studying the return type of functions in the LLVM back-end which deal with

compiling *CmmStmt's*. Statements just as expressions also all return the same basic type when compiled to LLVM code by the back-end. This type is shown below in *listing 3.11*.

```
1 type StmtData = (LlvmEnv, [LlvmStatement], [LlvmCmmTop])
```

Listing 3.11: LLVM Compilation type for a Cmm Statement

- `LlvmEnv`: As compiling a Cmm statement usually involves also compiling a Cmm expression, this LLVM Environment performs the same purpose of returning an updated environment if new external Cmm Label's have been encountered. This first case updates the environments global map, as a new global variable has been created as shown in *listing 3.8*. In the case of a *CmmStore* statement though, a Cmm local register may be encountered for the first time. It will be allocated on the stack and added to the local map of the environment. This is further explained in *section 3.3.3.3*.
- `LlvmStatements`: A *CmmStatment* is compiled to a list of LLVM Statements.
- `[LlvmCmmTop]`: Serves the same purpose as it does for Cmm expression code generation.

3.3.3.3 Handling LLVM's SSA Form

One of the main difference between *Cmm* and *LLVM Assembly* is the requirement that LLVM Assembly be in single static assignment form. This was outlined in *section 2.6.1*. Thankfully, this is actually quite easy to handle.

LLVM allows for data to be explicitly allocated on the stack, using its `alloca` instruction. This instruction provides an alternative to producing SSA formed code. If a mutable variable is needed, then it is allocated on the stack with `alloca`. The value returned from this instruction is a pointer to the stack memory and this memory location can be read from and written to just like any other memory location in LLVM by using the `load` and `store` instructions respectively. While this initially allocates all these variables on the stack and doesn't use any registers, LLVM includes an optimisation pass called *mem2reg* which is designed to correct this, changing explicit stack allocation into SSA form instead which can use machine registers when compiled to native code. This approach to handling LLVM's SSA form is in fact the method that the LLVM developers themselves recommend. Below in *listing 3.12* you can see the style of stack allocated code generated by the LLVM back-end and in *listing 3.13* you can see the result after the *mem2reg* optimisation has been applied.

```
1 declare i32 @getchar()
```

```
2
3  define i32 @main() {
4  entry:
5        %b = alloca i32
6        %a = alloca i32
7
8        %tmp1 = call i32 @getchar( ) nounwind
9        store i32 %tmp1, i32* %b
10
11       %tmp2 = load i32* %b
12       %toBool = icmp ne i32 %tmp2, 0
13       br i1 %toBool, label %bb, label %bb5
14
15 bb:
16       store i32 1, i32* %a
17       br label %bb6
18
19 bb5:
20       store i32 0, i32* %a
21       br label %bb6
22
23 bb6:
24       %retval = load i32* %a
25       ret i32 %retval
26 }
```

Listing 3.12: LLVM Stack Allocated code before *mem2reg* optimisation

```
1  declare i32 @getchar()
2
3  define i32 @main() {
4  entry:
5        %tmp1 = call i32 @getchar() nounwind
6        %toBool = icmp ne i32 %tmp1, 0
7        br i1 %toBool, label %bb, label %bb5
8
9  bb:
10       br label %bb6
11
12 bb5:
13       br label %bb6
14
```

```
15 bb6:
16         %a.0 = phi i32 [ 0, %bb5 ], [ 1, %bb ]
17         ret i32 %a.0
18 }
```

Listing 3.13: LLVM Stack Allocated code after *mem2reg* optimisation

3.3.4 Handling Registered Code

As mentioned before (see *section 3.3.1*) the major difference for back-end code generation for handling Cmm *registered* code as compared to *unregistered* is the use of pinned STG virtual registers and the TABLES_NEXT_TO_CODE optimisation.

To handle the TABLES_NEXT_TO_CODE optimisation, the LLVM back-end simply disables it. This can be done independent of enabling or disabling all of registered mode. While this is essentially a cop out, I decided that the amount of time and effort required to support this optimisation was not worth the performance gains. LLVM itself can fundamentally not support this feature, leaving us with they only option really of patching GHC's assembly optimisation pass to work with the assembly produced by LLVM so that they can rearrange the data to implement this optimisation. This is the approach taken by the C back-end and while it works, it has proven itself to be complex and involve a fair amount of maintenance. These are all areas in which the LLVM back-end hopes to improve over GHC's existing back-ends. Also, as can be seen in *section 4*, this optimisation no longer seems to offer the performance benefits it once did, with the resulting improvement being less then 1%.

Handling the STG virtual registers though, is something which can't be avoided. In the C and NCG back-ends, this optimisation alone offers a drastic improvement in the performance of the compiled code, on average halving the runtime (see *section 4*). The requirement of a back-end code generator is to place the STG virtual registers in specific hardware registers and guarantee they will be there when expected, such as when calling into the run-time-system. To achieve this the LLVM back-end uses a custom calling convention that passes the first n arguments of a function call in the specific registers that the STG registers should be pinned to. Then, whenever there is function call, then LLVM back-end generates a call with the correct STG virtual registers as the first n arguments to that call.

Why does this work? It works as it guarantees that on the entrance to any function, the STG registers are currently stored in the correct hardware registers. It also guarantees this on a function exit since all Cmm functions that GHC generates are exited by tail calls. In the function itself, the STG registers can be treated just like normal variables, read and written to at will. While the LLVM register allocator may not keep the STG registers fixed to the

hardware registers they are expected to be stored in; in fact it may spill them to the stack, this works fine. The semantics of pinning the STG virtual registers really only guarantees that they will be found in the correct hardware registers on a functions entrance and exit, which is what the custom calling convention guarantees. The fact that are permanently stored in these hardware registers, reserving them entirely, is just a result of the particular implementation technique used by the C and NCG back-ends, not a requirement. Interestingly this technique actually offers a performance advantage over the approach taken by the C and NCG back-end, as now the register allocator has more flexibility to produce optimal code since it can still make use the hardware registers that have been assigned to STG registers. If the register allocation for a function is to leave the STG registers in the hardware registers for the whole function, the approach taken by the LLVM back-end allows this. The register allocator should realise this is the optimal approach and produce code with this allocation. If however the optimal register allocation for a function differs from this and involves spilling the STG registers to the stack for example so that the registers they were assigned to can be used, then the LLVM back-end approach allows for the register allocator to produce such code while the C and NCG back-ends do not. This is especially the case in the most common architecture that Haskell code is compiled to, *x86*. This architecture only has five general purpose registers and GHC pins STG registers to four of them, leaving just one register free for use in a function. The performance impact of the approach taken by the LLVM back-end will be discussed in detail in *section 4*.

3.4 Other Stages

In this section we just briefly outline the stages in the LLVM back-end's pipeline which haven't been covered so far. Code is transferred between these stages by using temporary on disk files.

3.4.1 LLVM Assembler

This is a very simple stage in which the human readable text version of LLVM assembly code is translated to the binary bitcode format. This is done by simply invoking the LLVM `llvm-as` tool on the stage input file.

3.4.2 LLVM Optimisation

In this section a range of LLVM's optimisations are applied to the bitcode file, resulting in a new optimised bitcode file. This is done by simply invoking the LLVM `opt` tool on the stage

input file. The optimisations are selected using the standard optimisation groups of *'-O1', '-O2', '-O3'* provided by `opt`, depending on the level of optimisation requested by the user when they invoked GHC.

3.4.3 LLVM Compiler

This is the final stage in which the input LLVM bitcode file is compiled to native assembly for the target machine. This is done by simply invoking the LLVM `llc` tool on the stage input file.

4

Evaluation

In this section I will evaluate the new LLVM back-end, particularly in comparison to GHC's existing code generators, the C back-end and native code generator (*NCG*). This evaluation will be done on two primary dimensions, complexity of the back-end itself (see *section 4.1*) and the performance of the generated code (see *section 4.2*). The first issue is of primary concern for the developers of GHC, while the second is of concern to both the developers and users of GHC.

This evaluation will specifically look at the following:

Ease of initial implementation
How much work is required to implement the back-end, particularly in regard to how well it fits in with the existing compiler design. Ideally, the back-end should require little work to implement and require no changes to the existing compiler pipeline and Haskell execution model.

For this criteria, I expect the LLVM back-end to stand out in comparison to the C back-end and NCG. As outlined in *section 2.3.1* and *2.3.2* for the C back-end and NCG back-end respectively, this is an area where both have considerable problems.

Ease of future development
How much ongoing work is required to maintain and extend the back-end.

As with above, I expect that LLVM will perform well against this criteria. Particularly since one of the benefits of the LLVM back-end is that it outsources the work of code generation so that future developments in LLVM will directly benefit GHC at no cost. The expected lower complexity of the LLVM back-end also should be of help in this criteria.

Speed of compilation
The amount of time it takes for GHC to compile a Haskell program. Ideally as short as possible.

Currently, the C code generator is far slower then the NCG, on the order of two to three times slower. I believe that the LLVM back-end will fit in the middle of them, slightly closer to the NCG.

Quality of produced code
Is the quality of the produced machine code of suitable quality? Particularly in comparison to the other back-ends. The quality of the code can be measured on the basis of its runtime, and file size. Ideally we want both as small as possible.

Given the results of the EHC LLVM back-end (see *section 2.8.1*) and the aggressive optimisations performed by LLVM this is an area in which I expect LLVM to outperform the existing back-ends. I do not expect it to be of a similar level as that achieved by the EHC LLVM back-end though as GHC produces far more efficient code already then EHC does and any improvement at all to GHC requires a lot of effort because of the existing high standard.

4.1 Complexity of Implementation

In this section I will evaluate the complexity of the LLVM back-end in comparison to GHC's C and native code generator back-ends. Complexity can take many forms but we are interested primarily in the amount of ongoing maintenance required, and effort taken to implement new features. We are also interested in the initial effort that was required to implement each back-end but as GHC is a very mature project and all three back-end have already been implemented, this is of lesser concern.

4.1.1 Code Size

The first and easiest measure of the complexity of each of the back-ends, is their respective code size. This will give us a quick easy indication of the amount of work initially required

C	**Total**	3323
	Compiler	1122
	Includes	2201
NCG	**Total**	20570
	Shared	7777
	X86	5208
	SPARC	4243
	PowerPC	3342
LLVM	**Total**	3133
	Compiler	1865
	LLVM Module	1268

Table 4.1: GHC Back-end Code Sizes

to implement them, as well as the ease to extend them in the future. Below in *table 4.1* is a breakdown of the code size for the three back-ends.

As you can see from the table, the LLVM back-end is in fact the smallest in terms of lines of code. The C back-end is of fairly similar size, and a large part of it is actually header files which are included in the generated C code that define macros and type definitions used in the code compiled by the C back-end. It is also easy to see the distinct disadvantage of producing a native code generator. It is over 6 times the size of either the C or LLVM back-end, yet it only supports 3 architectures while the LLVM back-end theoretically supports 9 and the C back-end even more.

Code size isn't able to tell the whole story though, so in the next section we will look at the complexity in terms of its external dependencies.

4.1.2 External Complexity

While code generation is a large part of any back-end, there is also a surrounding infrastructure needed to then compile the produced code to an executable or object file. The complexity of this infrastructure can vary greatly between the back-ends.

The C back-end is the worst off here due to the deficiencies with using the C programming language as a compilation target. As discussed in *section 2.3*, the C back-end isn't natively able to support compiling *Cmm* code that is in the registered form. To achieve this it instead relies on post processing the assembly code produced by the C compiler, in this case *gcc*. This is no easy task and also very fragile. Other C compilers then *gcc* can not be easily used, a problem for the Windows platform on which *gcc* is not well supported. Also as different versions of *gcc* produce slightly different assembly code, then the post processing infrastructure needs to be continually updated for it to work with new versions of *gcc*. This is all quite a nightmare and has caused considerable pains for the GHC developers.

The native code generator is much better off in this regard. As it gives GHC a complete compiler pipeline, producing assembly code, there is very little in the way of external infrastructure required. Indeed all that is needed is an assembler, which are well supported an easily found on all platforms.

Finally, then this brings us to the new LLVM back-end. The LLVM back-end lies somewhere in the middle between the lack of external complexity represented by the NCG and the extreme pains of the C back-end. For compiling unregistered code, the LLVM back-end just requires the use of a standard LLVM installation. As LLVM is easily found and well supported on all platforms that GHC currently runs on, this is not an issue. However, the problem is when the LLVM back-end wants to compiler registered code, which in reality is a requirement due to the large performance gains. As we discussed in *section 3.3.4*, to handle registered code the LLVM back-end needs to use a custom calling convention. Unfortunately implementing a custom calling convention in LLVM requires that the source code itself be modified and LLVM rebuilt. Thus, the LLVM back-end is dependent not on LLVM itself, but a customised version, specific to GHC. The changes required to LLVM itself are quite small and easily implemented and maintained, the main issue is that the custom LLVM build would also need to be distributed with GHC. This however doesn't affect the actual complexity of the back-end itself, it just creates a slight problem of increasing GHC's distribution and installation size a noticeable amount.

4.1.3 State of the LLVM back-end

One strong indication of the ease of use of LLVM and the complexity of the LLVM back-end for GHC is the state of the back-end. Development of the back-end was only begun 3 months ago, yet in that time it has reached two major milestones:

- ***Compiles Unregistered Code***: The first milestone reached by the LLVM back-end was the ability to compile Haskell code in unregistered mode. This was reached after only 7 weeks of development. By this stage the LLVM back-end was able to pass the majority of the GHC test suite.
- ***Compiles Registered Code***: This milestone was reached after a further 3 weeks of development, with not all the time spent on this aim.

4.2 Performance of LLVM back-end

In this section we will look at the performance of the LLVM back-end. This will primarily be measured by studying the run-time of the code compiled via the LLVM back-end in comparison

to GHC's C and NCG back-ends. We will also look at compilation times and the size of the compiled code.

To measure these dimensions I have used two benchmark suites for Haskell code, they are as detailed below.

- ***NoFib*** [28]: The standard benchmark suite for Haskell and GHC. It is distributed with GHC itself and regularly used to check the performance of new additions to GHC and test for any regressions. It includes three groups of programs, *imaginary*, *spectral* and *real*. The *imaginary* programs are fairly small programs that don't represent typical Haskell code, however they are useful for showing up performance bugs. The *spectral* programs consists of core algorithms you might find in real programs, such as fast Fourier transforms. They represent somewhat realistic programs while still being of modest size. Finally, the *real* programs are real Haskell programs, they are programs written by people wanting to get things done and have later been included in *nofib*.
- ***Data Parallel Haskell*** [9]: This is an in-progress extension and library for GHC and Haskell which enables support for nested data parallelism. As part of this work the authors have implemented a benchmark suite which tests the performance of parallel Haskell code. This is useful for benchmarking the back-end as the produced code contains a lot of tight loops and small areas of code which are frequently executed. This type of code places a lot of strain on the register allocator and optimisations, which should help show up the advantages of LLVM in this area.

We will evaluate the performance of the LLVM back-end for both *unregistered* and *registered* code, although the focus will be mostly on registered code. GHC is nearly always used in registered mode due to the performance gains it offers over unregistered code. The main use of unregistered mode in GHC is for portability purposes. As we discussed in *section 3.3.1* compiling code in registered mode is far more difficult and as such, less portable. The amount of work required to get GHC running on a new platform in unregistered mode is far less then getting it running in registered mode.

All of the benchmarks were run on a Pentium Core2 dual core processor, running at 2.40Ghz, with 3.4GB of memory and on a Linux 2.6.28 kernel.

4.2.1 Unregistered results

For evaluating the performance of the LLVM back-end in unregistered mode, the *nofib* benchmark suite was ran against both the LLVM back-end and the C back-end. The native code

LLVM back-end against C back-end (Δ%)		
Program	Code size	Runtime
anna	+6.8	-13.1
atom	+0.1	-11.5
cacheprof	+2.8	-51.1
cichelli	+0.3	-21.6
circsim	+0.5	-17.7
compress	+0.4	-62.6
constraints	+0.3	-4.8
cryptarithm1	+0.0	-17.9
exp3_8	+0.1	-7.5
listcopy	+0.2	+14.8
mandel	+0.2	-9.5
para	+0.5	-59.0
puzzle	+0.3	-21.4
simple	+3.1	-16.3
solid	+0.8	-22.7
typecheck	+0.4	-13.3
wave4main	+0.3	-28.3
(74 more)	..	..
Min	+0.0	-62.6
Max	+6.8	+14.8
Geometric Mean	+0.7	-15.9

Table 4.2: nofib: Unregistered Performance

generator is not capable of running in unregistered mode so it was excluded from the comparison. The results are summarised below in *table 4.2*.

As can be clearly seen from these results, the LLVM back-end offers considerable performance advantages over the C back-end and produces largely the same size code. In some cases such as the performance of the `compress` nofib program, these performance improvements allow the unregistered mode code to approach near the levels of registered code. Despite this performance improvement, there is also a slight decrease in the compile times with the LLVM back-end compared to the C back-end. These can be seen below in *table 4.3*.

Clearly the LLVM back-end does a better a job then the C back-end at handling unregistered code. It manages to provide an average of 16% improvement to the run-time of the code, while keeping both code size and compilation time at a similar level as the C back-end. It's interesting to see that this level of improvement is somewhat similar to the one achieved by the EHC LLVM back-end. While it did only achieve 11% as opposed to the 16% improvement in runtime gained with GHC, this was using version 2.3 of LLVM while the GHC LLVM back-end is using version 2.5.

The more important concern though is the performance of the LLVM back-end when compiling

LLVM back-end against C back-end (Δ%)	
Program	Compile time
ansi	-4.0
atom	+0.0
awards	-16.7
banner	-2.9
bernouilli	-1.6
boyer	+0.6
calendar	-4.5
genfft	+0.0
life	-4.1
parstof	-10.4
simple	-3.2
treejoin	-4.5
x2n1	-22.6
(84 more)	..
-1 s.d.	-27.4
+1 s.d.	+31.5
Average	-2.3

Table 4.3: nofib: Unregistered compile times

registered code. We will now look at this in the next section.

4.2.2 Registered Results

When compiling Haskell code in registered mode, GHC offers two existing back-ends, the native code generator and the C back-end. As part of this thesis I have implemented a third option, the LLVM back-end. We will now look at the performance of this back-end by studying the results of the *nofib*, and *data parallel haskell* benchmark suites.

4.2.2.1 Nofib Results

Firstly, you can see the run-time results gained from the *nofib* benchmark suite below in *table 4.4*. This table shows three columns, the first is simply the test programs name. The third column shows the runtime of the test program when compiled via the LLVM back-end as a delta of the runtime of the performance of the test program when compiled via the NCG back-end. A larger positive value indicates a faster runtime when the program is compiled with the NCG back-end and a larger negative value indicates a faster run-time when compiled via the LLVM back-end. The second column shows the same information as the third but for the C back-end compared with the NCG back-end.

LLVM and C back-end against NCG back-end (Δ%)		
Program	C Runtime	LLVM Runtime
atom	2.3	9.2
bernouilli	-0.9	-0.9
cacheprof	5.6	4.6
comp_lab_zift	5.7	2.8
constraints	5.5	2.9
cryptarithm1	26.4	6.4
fulsom	4.7	7.7
hidden	1.1	5.8
integer	2.5	3.0
integrate	11.1	-1.9
lcss	1.9	1.9
life	4.2	0.8
multiplier	0.12	0.12
para	1.4	9.6
power	3.4	0.4
scs	0.7	2.4
transform	12.1	5.8
treejoin	7.1	2.7
typecheck	7.9	13.2
wheel-sieve2	7.2	4.0
(71 more)	..	..
-1 s.d.	-2.4	0.4
+1 s.d.	17.0	7.2
Average	6.9	3.8

Table 4.4: nofib: Registered Performance

The native code generator offers the best performance of the three but its a fairly close race, with the LLVM back-end less then 4% percent behind and the C code generator less then 7%. While this is slightly disappointing that LLVM doesn't provide a performance advantage over the NCG back-end, it is very promising given its early stage of development that it is able to produce code of the same level of quality as the NCG and C back-end. Both of these back-ends benefit from years of work and optimisation, while the new LLVM back-end has only been in development for around 3 months and only just recently reached a usable state. The *nofib* benchmark suite provides other useful performance metrics such as the run-time excluding garbage collection (known as *mutator time*), the time for garbage collection, code size and compilation times. These are summarised below in *table 4.5*, the average deltas of the C code generator against the NCG are presented in column two, with the LLVM code generator against the NCG presented in column 3.

As can be seen from this table, the LLVM code generator produces slightly large binaries then either the NCG or C code generator. Some of this appears to be due to optimisations it implements such as in-lining, which directly increase the size of the code. Allocations for

LLVM and C back-end against NCG back-end (Δ%)		
Metric	C Back-end	LLVM Back-end
Binary Sizes	+0.8	+7.0
Allocations	+0.0	+0.0
Mutator Times	+8.1	+6.8
GC Times	+0.4	+1.5
Compilation Times	+312.3	+149.7

Table 4.5: nofib: Registered Performance Breakdown

all back-ends is equal as this is not a property that the code generators change. Both the Mutator time and GC time are slightly higher for the LLVM back-end compared with the NCG back-end. This is to be expected though as it is in-line with the increase in the overall run-time and doesn't indicate any performance bug in these areas. The possible reasons for the slight performance loss will be examined in *section 4.2.3*.

4.2.2.2 Data Parallel Haskell Results

In this section we will further evaluate the LLVM back-end in registered mode against GHC's native code generator. For this purpose we will use the *Data Parallel Haskell* benchmark suite, since as mentioned before the native code which is generated from this benchmark suite puts a lot of pressure on the register allocator and optimisations available in the back-ends.

The results from running the benchmark against both the NCG and LLVM back-end are below in *table 4.6*. The C back-end was not included due problems using the Data Parallel library and extension with it. The final column shows the runtime of the benchmark programs when compiled via the LLVM back-end, as a percentage of the runtime when compiled via the NCG back-end. A larger negative value indicates a faster program when compiled via LLVM.

As can be seen, the LLVM back-end shows a very impressive performance gain for all the tests, bringing an average reduction of 25% in the runtime of the tests. The reasons for this are two fold. Firstly the LLVM optimiser and register allocator simply outperform the NCG. Secondly and of far greater impact is the use of the custom calling convention used to implement registered mode, as outlined in *section 3.3.4*. Using a calling convention to pass the STG registers in the correct hardware registers allows far more room to optimise the code and generate the most appropriate register allocation then the method taken by the NCG of permanently fixing the STG registers to their corresponding hardware registers. On an x86 machine, this means that there is only one general purpose register free for the NCG register allocator to use, all else must be spilled to the stack. In many situations this works fine as GHC uses the STG virtual registers for the vast majority of work. However in the case of the DPH benchmark the code generator is forced to do a lot of work, with a lot of tight code loops involved that need to use

LLVM against NCG back-end (Δ%)			
Program	Mode	Threads	Runtime
SumSq			
	primitives	Sequential	-14.3
	vectorised	Sequential	-7.9
	primitives	1	-44.2
	primitives	2	-48.1
	vectorised	1	-44.0
	vectorised	2	-50.0
DotP			
	primitives	Sequential	-24.1
	vectorised	Sequential	-23.6
	primitives	1	-46.2
	primitives	2	-22.7
	vectorised	1	-17.4
	vectorised	2	-24.2
SMVM			
	primitives	Sequential	-20.4
	vectorised	Sequential	-18.6
	primitives	1	-6.7
	primitives	2	-13.1
	vectorised	1	-9.7
	vectorised	2	-11.5
Average			-25.0

Table 4.6: DPH: Performance

multiple registers. Given the results above I believe that the GHC native code generator could also benefit from the using a custom calling convention to implement the STG virtual registers, instead of fixing them.

4.2.3 Performance of LLVM back-end Examined

As we saw in *table 4.4*, the LLVM back-end for the *nofib* benchmark produces slightly slower code in general then the NCG. Surprisingly, it also never outperforms the NCG in any of the *nofib* benchmarks. What could be the reason for this? There seems to be three, firstly the LLVM optimiser has no impact on the runtime of the code produced by the GHC LLVM back-end. Secondly the LLVM compiler produces unnecessary stack manipulation; and finally, there is a slight performance loss from the disabling of the *TABLES_NEXT_TO_CODE* optimisation.

Below in *table 4.7* is a comparison of the runtime of *nofib* when compiled with the LLVM optimiser disabled, against when it is enabled. The column shows the runtime of the *nofib* benchmark suite when compiled without any LLVM optimisations, as a percentage of the

runtime when compiled with optimisations. A negative value indicates the program runs faster without optimisations.

Optimisations against No Optimisations	
Program	Runtime
atom	-4.7
bernouilli	+0.4
constraints	-0.7
cryptarithm1	-3.0
event	0.17
fulsom	+2.0
hidden	-2.3
integer	-0.2
integrate	+5.2
lcss	+1.2
para	-2.2
power	+0.6
primetest	+3.1
scs	-1.5
simple	-2.2
transform	-2.7
treejoin	-4.1
typecheck	+0.8
wave4main	+3.0
wheel-sieve1	+1.4
(71 more)	...
Min	-4.8
Max	+5.2
Geometric Mean	-0.5

Table 4.7: nofib: Effects of LLVM Optimiser

As can be seen from the table, the LLVM optimiser produces no noticeable effect on the runtime of the program, indeed for the *nofib* benchmark suite it actually slowed it down.The reasons for this are not clear but I believe that a large part of it is because the LLVM compiler suite has largely been tested against and optimised for procedural programming languages. One of the design goals of the LLVM Assembly language was to expose enough information about the nature and purpose of the code to the LLVM compiler and optimiser, while still retaining as low level nature as possible. Unfortunately the code produced by GHC is very difficult for LLVM to optimise, as a lot of information is hidden from it because of GHC's use of the STG virtual machine, which implements the actual semantics of Haskell, not the *Cmm* code being compiled. As STG defines its own stack and heap, manipulating them itself, this kind of information is no longer available to LLVM.

The second issue with the code produced by the LLVM back-end is that it contains a large amount of instructions which unnecessarily manipulate the hardware stack. For example, in

listing 4.1 below we can see a typical assembly function produced by the LLVM back-end. The code produced by LLVM on entry and exit manipulates the stack pointer, despite the stack not being used in the function at all.

```
1  Utils_doStatefulOp1_entry:
2          subl   $4, %esp    # Redundant instruction
3          movl   4(%ebp), %eax
4          movl   8(%ebp), %ecx
5          movl   (%ebp), %esi
6          movl   %eax, 8(%ebp)
7          movl   %ecx, 4(%ebp)
8          addl   $4, %ebp
9          addl   $4, %esp    # Redundant instruction
10         jmp    stg_ap_pp_fast
```

Listing 4.1: Inefficient code produced by LLVM

This example is only one of the simpler ones. Unfortunately there are many other cases in which LLVM not only unnecessarily manipulates the stack but also seems to produce an inefficient register allocation such that the stack is used for variable storage when it need not be. This seems to be the primary difference in the code produced by the NCG and LLVM. The LLVM optimiser appears to further exacerbate this issue, with the optimiser often introducing further inefficient stack operations. For example in some cases the optimiser produces codes code that stores a computed value on the stack if it needs to be used again, when it would be actually far quicker to recompute it, or in some cases a register allocation exists which doesn't require storing it on the stack.

4.2.3.1 Performance Impact of TABLES_NEXT_TO_CODE

One of the optimisations used by GHC that I was unable to support in the LLVM back-end was the so called TABLES_NEXT_TO_CODE optimisation. Please see *section 2.5* for a description. To determine what impact this had on the performance of the LLVM back-end, I ran the *nofib* benchmark suite against the NCG back-end, once with the optimisation enable and a second time with it disabled. The results are presented below in *table 4.8*. A greater positive value indicates a larger runtime with the LLVM optimisations disabled.

It seems that this optimisation has little impact on the run time of *nofib*, although even this small amounts does account for some of the loss in performance in the LLVM back-end. Also of interest though is the use of keeping this optimisation in GHC itself? Given the difficulty it would cause the LLVM back-end to implement, there is no reason to do so. In the case of the C

Impact of TABLES_NEXT_TO_CODE (Δ%)	
Program	Runtime
atom	+1.5
cacheprof	+1.9
fulsom	+2.5
integrate	+3.0
para	-2.5
primetest	+3.7
treejoin	+3.4
typecheck	+3.4
wheel-sieve1	-2.9
(82 more)	...
Min	-2.9
Max	+4.2
Geometric Mean	+0.8

Table 4.8: nofib: Effects of TABLES_NEXT_TO_CODE Optimisation

back-end, this optimisation also causes considerable problems and requires that the assembly code produced by *gcc* after compiling the C code be processed by an assembly level optimiser, which is tricky and tiresome to maintain.

4.3 Summary

Now that we have looked at the complexity and performance of the LLVM back-end in regards to GHC's C and NCG back-ends it is useful to summarise the findings and relate them back to our evaluation criteria introduced in *section 4*.

Ease of initial implementation
LLVM proved itself to be a very straight forward and easy compiler to target. In only a short development period the LLVM back-end has gone from scratch to being able to pass the majority of the GHC test suite and is capable of building the vast majority of GHC itself. While there were some challenges involved in getting the LLVM back-end working with registered code, the custom calling convention proved to be an ideal solution that was implemented quite easily. The outcome here if anything exceeded the expectations.

Ease of future development
I believe that this is one of LLVM's strongest areas. Firstly, the LLVM back-end itself is quite small and self contained, making it easy to understand and modify. Also, through the LLVM back-end GHC gains access to an entire compiler infrastructure. Back-end optimisations for GHC can now be implemented as part of the

LLVM optimiser infrastructure, providing a strong platform for such work. LLVM itself will also continue to be developed and improved which in turn will benefit GHC, yet requires no work from the GHC developers.

Speed of compilation

Unfortunately the speed of compilation was slightly slower then was expected. While the LLVM back-end is twice as fast as the C back-end, the NCG is itself over twice as fast over the LLVM code generator. It should be fairly straight forward improving the LLVM back-end here though, it can benefit from both some optimisations to its Haskell code, as well as linking with the LLVM API so it can remove the LLVM assembler stage, generating bitcode directly instead. This was previously discussed in *section 3.2*.

Quality of produced code

For this criteria it is difficult to draw a general conclusion. The *nofib* benchmark showed that for registered code the LLVM back-end produced code which was slightly slower then the NCG but slightly faster then the C code generator. While for unregistered code the LLVM back-end produced far more efficient code then the C back-end. For the Data Parallel Haskell benchmarks LLVM produced tremendous results, averaging a 25% reduction in run time. A large part of this though seems to be the use of the custom calling convention to implement STG virtual registers as opposed to the use of LLVM itself. Finally, when looking at the code produced by LLVM in detail, it was found that the despite LLVM's impressive array of optimisation passes, they had no impact on the code; and LLVM's code generator was producing inefficient code that unnecessarily manipulated the hardware stack.

It would appear then overall that currently, the LLVM back-end produces fairly equal quality code to the NCG and C back-end. While I was expecting and hoping for a clear improvement across the board, this result is still a good indication of LLVM's suitability as a target for GHC. Given the amount of development time that has currently been invested in the NCG and C back-end, for the LLVM back-end to be able to match them in such a short time clearly demonstrates its effectiveness. The DPH benchmarks also demonstrate the potential for improvement that the LLVM back-end has.

5

Conclusion

This thesis describes the design and implementation of a new back-end for the Glasgow Haskell Compiler which uses the Low Level Virtual Machine compiler for code generation. It sets out to answer the question, *is the Low Level Virtual Machine compiler a suitable back-end target for the Glasgow Haskell Compiler?* This question was further broken down into the following question:

- Is the implementation reasonably straight forward. How does it compare in complexity to GHC's other back-ends?
- Is the generated code of similar quality to GHC's other back-ends?
- Which optimisations no longer apply and what new optimisations are now possible with LLVM?

After introducing the background material required to understand this thesis and then describing the design of the LLVM back-end, the back-end was evaluated in regards to these questions. The evaluation looked at the back-end from two broad dimensions, the complexity of the back-end design, which corresponds to the first question above, and the performance of the generated code, which corresponds to the second question above.

It was found that in regards to the implementation complexity, the LLVM back-end has the lowest complexity of all three of the back-ends. Unlike the NCG it has a far smaller code

base that doesn't need to handle issues such as position independent code. It is also portable, supporting all the platforms that GHC currently supports and is easily able to support others, as long as LLVM is supported on the desired platform. While the C back-end has a similar and perhaps even smaller code base, it requires post processing of the assembly produced by *GCC* to be able to produce efficient code. This ties GHC specifically to *GCC* and also causes a lot of maintenance issues for GHC. The LLVM back-end was not without fault though, since in order to support registered code it requires a custom calling convention, which can only be added to LLVM by changing its source code. Because of this, the LLVM back-end needs to also distribute with it a custom version of LLVM which adds to the distribution size and maintenance issues. These however are far more easily handled then those of the NCG or C back-end.

In regards to performance, all three back-ends were reasonably similar, with the LLVM back-end outperforming the C code generator in all regards, but having slightly lower performance against the native code generator with the *nofib* benchmark. This was only a value of 4% though of which around 1% was lost due to the disabling of the TABLES_NEXT_TO_CODE. As this optimisation has such a small impact on the runtime of Haskell code, I believe that it may be best to disable it for the C back-end as well since the C back-end, like the LLVM back-end, can't directly support this optimisation. It has to use fairly extreme methods to get around this limitation of C, implementing fragile assembly level optimisations. Finally, the use of the custom calling convention to implement the STG virtual registers worked out extremely well. Not only does it allow LLVM to produce code which is compatible with the code produced by the NCG or C back-ends, it also provides a performance advantage as we saw in the *DPH* benchmarks due to the extra registers and flexibility it allows the registers allocator. The results of the *DPH* benchmark of an average of a 25% reduction in runtime while an extreme case do show the benefits of the LLVM back-end and the improvements possible in the future.

There are some issues with the LLVM back-end though, the most troubling being the redundant stack manipulation code it generates. I am currently unsure of the cause of this but believe that it is one of three possibilities, or a combination of them. Firstly there is the possibility that the GHC back-end is producing inefficient LLVM code which triggers this performance bug, the prime suspect being some incorrect alignment directives. Secondly, there is a chance that the changes I made to LLVM to implement the new custom calling convention introduced this bug, especially since the changes involved altering register definitions. However I believe that the third possibility, that this is a performance bug in LLVM itself to be the case. As I discussed in *section 4.2.3*, the LLVM optimiser brings no improvement to the runtime of Haskell code and can in some cases decrease the performance. One of the reasons for cases when a decrease occurs is that the LLVM optimiser introduces more unnecessary stack manipulation. This would appear to confirm that the performance bug lies with LLVM itself. The lack of improvement to the code by the LLVM optimiser is one of the other main issues with the LLVM back-end and quite disappointing. Given the LLVM optimiser is one of LLVM's most publicised

features, its current complete uselessness is surprising.

In closing, I believe that LLVM offers a compelling and viable back-end target for GHC today. With far lower complexity, comparable performance and access to the rest of the LLVM infrastructure, it is an attractive alternative to the C and native code generator back-ends. Going forward I believe it is the most appropriate primary back-end target for GHC as the performance should be fairly easy to improve, giving it this edge over the other back-ends, while the amount of work required to maintain and extend it is minimal.

Bibliography

[1] clang: A C language family frontend for LLVM. *See* `http://clang.llvm.org/`.

[2] LDC: LLVM D Compiler. *See* `http://www.dsource.org/projects/ldc`.

[3] llvm-lua, JIT/Static compiler for Lua using LLVM on the backend. *See* `http://code.google.com/p/llvm-lua/`.

[4] MacRuby. *See* `http://www.macruby.org/`.

[5] OpenJDK - Zero-Assembler Project. *See* `http://openjdk.java.net/projects/zero/`.

[6] The GHC Commentary. *See* `http://hackage.haskell.org/trac/ghc/wiki/Commentary`.

[7] Unladen Swallow: A faster implementation of Python. *See* `http://code.google.com/p/unladen-swallow/`.

[8] *IEEE Standard for Floating-Point Arithmetic*, August 2008.

[9] M. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. Data parallel haskell. *See* `http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`.

[10] GHC Developers. The Glasgow Haskell Compiler. *See* `http://www.haskell.org/ghc/`.

[11] GHC Developers. The Glasgow Haskell Compiler Source Code. *See* `http://darcs.haskell.org/ghc/`.

[12] A. Dijkstra, J. Fokker, and S. Doaitse Swierstra. The structure of the Essential Haskell Compiler or coping with compiler complexity. *IFL '07: Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, pages 107–122, 2007.

[13] Free Software Foundation. GNU GCC. *See* `http://gcc.gnu.org/`.

[14] Free Software Foundation. *GNU GCC Manual*, 4.3.2 edition, August 2008. *See* `http://gcc.gnu.org/`.

[15] Albert Graf. The Pure Programming Language. *See* `http://code.google.com/p/pure-lang/`.

[16] F. Henderson, T. Conway, and Z. Somogyi. Compiling logic programs to c using gnu c as a portable assembler. 1995.

[17] Apple Inc. OpenCL on the Mac Platform. *See* `http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCLontheMacPlatform/OpenCLontheMacPlatform.html`.

[18] ECMA International. *Standard ECMA-355 - Common Language Infrastructure (CLI). Technical Report 4th Edition.* ECMA International, June 2006. *See* `http://www.ecma-international.org/publications/standards/ecma-355.htm`.

[19] S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, pages 127–202, 1992.

[20] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The glasgow haskell compiler: a technical overview. December 1992.

[21] S. Peyton Jones, N. Ramsey, and F. Reig. C–: a portable assembly language that supports garbage collection. 1999.

[22] C. Lattner and V. Adve. *LLVM Language Reference Manual.* *See* `http://llvm.org/docs/LangRef.htm`.

[23] C. Lattner and G. Henriksen. Accurate Garbage Collection with LLVM. *See* `http://llvm.org/docs/GarbageCollection.html`.

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

[25] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification Second Edition.* Prentice Hall PTR, April 1999. *See* `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`.

[26] S. Marlow and S. Peyton Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 4–15, New York, NY, USA, 2004. ACM.

[27] B. O'Sullivan and L. Augustsson. llvm: Bindings to the llvm compiler toolkit. *See* `http://hackage.haskell.org/package/llvm`.

[28] W. Partain. The nofib benchmark suite of haskell programs. pages 195–202, London, UK, 1993. Springer-Verlag.

[29] N. Ramsey, J. Dias, and S. Peyton Jones. Hoopl: Dataflow optimization made simple. *Submitted* to the 2010 ACM Symposium on Principles of Programming Languages, 2009. *See* `http://www.cs.tufts.edu/ nr/pubs/dfopt-abstract.html`.

[30] D. Tarditi, P. Lee, and A. Acharya. No assembly required: Compiling standard ml to c. *ACM Letters on Programming Languages and Systems*, 1990.

[31] The LLVM Team. *LLVM: Frequently Asked Questions. See* `http://llvm.org/docs/FAQ.html`.

[32] John van Schie. Compiling Haskell To LLVM. Master's thesis, 2008.