An LLVM Backend for GHC

David A. Terei Manuel M. T. Chakravarty

University of New South Wales

{davidt,chak}@cse.unsw.edu.au

Abstract

In the presence of ever-changing computer architectures, high-quality optimising compiler backends are moving targets that require specialist knowledge and sophisticated algorithms. In this paper, we explore a new backend for the Glasgow Haskell Compiler (GHC) that leverages the *Low Level Virtual Machine (LLVM),* a new breed of compiler written explicitly for use by other compiler writers, not high-level programmers, that promises to enable outsourcing of low-level and architecture-dependent aspects of code generation. We discuss the conceptual challenges and our backend design. We also provide an extensive quantitative evaluation of the performance of the backend and of the code it produces.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.4 [*Processors*]: Code generation, Retargetable compilers

General Terms Design, Languages, Performance

1. Introduction

The Glasgow Haskell Compiler (GHC) began with a backend translating code for the *Spineless Tagless G-machine (STG-machine)* to C [23]. The idea was that targeting C would make the compiler portable due to the ubiquity of C compilers. At the time, this was a popular approach [7, 14, 27]. By leveraging C as a portable assembly language, the authors of compilers for higher-level languages hoped to save development effort, reuse the engineering work invested into the backend of optimising C compilers, and achieve portability across multiple architectures and operating systems.

Unfortunately, it turned out that C is a less than ideal intermediate language, especially for compilers of lazy functional languages with their non-standard control flow [24]. In particular, C does not support proper tail calls, first-class labels, access to the runtime stack for garbage collection, and many other desirable features. This is not surprising, as C was never designed to act as an intermediate language for high-level compilers. Nevertheless, it complicates the work of the compiler writers, as they have to work around those limitations of C-based backends. Moreover, the resulting machine code is less efficient than that of backends which generate native assembly code. GHC and other high-level compilers partially mitigate these shortcomings by targeting the GNU C compiler and using some of its many language extensions, such as global registers and first-class labels. This doesn't detract too much from the portability of C as GNU C itself has been ported to many architectures. However, exploiting GNU C extensions only partially solves the problems of compiling via C, as C compiler optimisations are often ineffective for code generated from high-level languages — much static information gets lost in the translation. Since the exact set of supported extensions depends on the particular version of GNU C, this approach also introduces new dependencies.

In the case of GHC, the use of the GNU C compiler as a backend also increases compilation time significantly. As a response, GHC eventually included support for native code generators that directly produce assembly code. These are currently only fully functional for the x86 and the SPARC architecture.

The desire to reap the benefits of compiling via C, while avoiding the problems, inspired the development of low-level intermediate languages that can be conveniently targeted by high-level compilers and that provide the basis for code generators shared by multiple compilers. Of particular interest is C--, as its design has been heavily influenced by the experience with GHC [24].

Although using C-- as an intermediate language is technically a very promising approach, it comes with a huge practical problem: it is only worthwhile to develop a portable compiler backend if it is used by many compilers, but compiler writers do not want to commit to a backend technology unless they know it is widely used and supported. As a consequence, a variant of C-- is currently used as a low-level intermediate language in GHC, but there is no useful general-purpose backend based on C-- that GHC could target.

Currently, the most promising backend framework is the *Low Level Virtual Machine (LLVM),* which comes with support for just-in-time compilation and life-long program analysis and optimisation [19]. An LLVM-based C compiler, named *clang*, recently gained significant traction as an alternative to the GNU C compiler.[1] Hence, it is very likely that LLVM will be further developed and is a suitable target of a long-term strategy.

In this paper, we describe the design of a new GHC backend that leverages LLVM. We illustrate the problems that we encountered, such as conflicting register conventions and GHC's tables-next-to-code optimisation, and our approach to solving them. We also present a quantitative analysis of both the performance of the backend itself and of the code it produces. In particular, we compare it to the C backend and the native code generator of GHC. The overall outcome is very promising: the new LLVM backend matches the performance of the existing backends on most code and outperforms the existing backends by up to a factor of 2.8 on tight loops with high register pressure on the x86 architecture.

In summary, we make the following technical contributions:

- We qualitatively compare GHC's existing backends and the capabilities of the LLVM framework (Sections 3 & 4).
- We present a design for an LLVM backend for GHC, including new methods to solve long standing problems, such as GHC's

[1] To a large part due to the backing and financial support of Apple.

fixed register assignment and tables-next-to-code optimisation (Section 5).

- We present a quantitative analysis of the performance of the two old backends and our new LLVM backend (Section 6).

We discuss related work in Section 7 and conclude in Section 8.

2. The case for a new backend

As we outlined in the previous section, GHC used to have two backends: (1) the *C backend,* which generates GNU C code, and (2) the *native code generator,* which generates assembly code for a few architectures. We will briefly review these two backends before giving our motivation for developing a third, the *LLVM backend.*

2.1 The C backend

The C backend is based on the *STG-machine*, an abstract machine that was designed to support the compilation of higher-order, lazy functional languages [23]. GHC's C backend generates C code which contains extensions that are specific to the GNU C compiler. These extensions facilitate the generation of more efficient code by storing the virtual machine registers of the STG-machine in concrete registers of the target hardware, by supporting tail calls with first-class labels, and by removing some overhead due to removing superfluous function entry and exit code. The resulting dependence on GNU C has two major drawbacks: Firstly, portability is limited by the portability of GNU C (it is not very well supported on Microsoft Windows, for example), and even on architectures that are supported by GNU C, the generated code can be of poor quality – as, for example, the code produced by the GNU C backend for the SPARC architecture. Secondly, as GHC not only exploits the extensions, but also the particular form of assembly generated, there are also dependencies on the version of the C compiler. Therefore, changes in the code generator of the GNU C compiler can break, and have in the past broken, GHC.

The GHC C code generator consists of a reasonably manageable 5,400 lines of code. However, to achieve even better efficiency than possible with only exploiting GNU C extensions, GHC opts to post-process the assembly generated by the C compiler. More precisely, it uses a Perl script to match specific patterns of assembly code and to rewrite them to better-optimised assembly code. This script is of course heavily dependent on the target architecture and also on the specific version of GNU C. In particular, it rearranges code blocks to implement the *tables-next-to-code* scheme of GHC, which we will discuss on more detail in Section 3.5. For obvious reasons, this script is hard to maintain (it is not a coincidence it is nicknamed "the evil mangler"), and has been responsible for more than one tricky bug.

Finally, another serious shortcoming of the C backend is its relatively long compilation time. GHC generates sizeable C files and GNU C requires considerable time to turn them into assembly code. Unfortunately, the long compilation time does not result in carefully optimised code, as one might hope. On the contrary, the generated assembly leaves much to be desired. This is not so much the fault of the GNU C compiler as a consequence of GHC generating non-idiomatic C code.

2.2 The native code generator

GHC's *native code generator (NCG)* was developed to avoid the problems of the C backend. It directly generates assembly code. Just as with C code generation, imperative code is generated from a representation of a Haskell program in the language of the STG-machine. As a result, and because appropriate care is taken, code generated by the NCG is binary compatible with code generated by the C backend. GHC's native code generator shares the usual advantages and disadvantages of a backend that produces assembly. It can generate efficient code, without any of the tricks used by the C backend, such as post-processing the assembly. Implementing a NCG, however, requires detailed knowledge of the target architecture and a considerable amount of work. Much of this work is replicated for each platform GHC supports. It is also quite difficult to implement code generators that generate high-quality code, as this requires many optimisations and fine tuning for each architecture to achieve optimal register and instruction selection. With the NCG, each platform, such as x86, SPARC, and PowerPC has to perform their own code generation with only the register allocator being shared among all platforms. As a result, the NCG, which includes three code generators (for x86, SPARC, and PowerPC) is at about 20,570 lines of code nearly four times the size of the C backend.

One of the main advantages of the NCG is that it can generally compile a Haskell program in half the time of the C backend. Also, despite its far larger size compared to the C backend, it is arguably the simpler of the two.

2.3 The LLVM backend

A GHC backend on the basis of a portable compiler framework, such as the *Low Level Virtual Machine (LLVM)* [19] , promises to combine the benefits of the C backend and the NCG with few or none of their disadvantages. The idea behind the C backend was to outsource the considerable development and maintenance effort required to implement a compiler backend to the developers of C compilers — after all, this is an area that is fairly far away from where GHC innovates.

Compared to the NCG and C backend, an LLVM has the following to offer:

Offloading of work. Building a high performance compiler backend is a huge amount of work, LLVM for example was started around 10 years ago. Going forward, GHC's LLVM backend should be a lot less work to maintain and extend than either the C backend or NCG. It will also benefit from any future improvements to LLVM which has a particularly bright looking future given the community and industrial support behind it [28].

Optimisation passes. GHC does a great job of producing fast Haskell programs. However, there are a large number of lower level optimisations, particularly the kind that require machine specific knowledge, that it doesn't currently implement. Some examples of these include partial redundancy elimination (PRE), loop unrolling and strength reduction. Through LLVM we gain these and many more for free.

The LLVM Framework. Perhaps the most appealing feature of LLVM is that it has been designed from the start to be a compiler framework. Individual optimisations can be chosen and ordered at compile time, as well as new optimisation passes dynamically loaded. LLVM also offers a choice of register allocators and even an interpreter and JIT compiler. For a research driven project like GHC this is a great benefit and makes LLVM a very fun and useful tool to experiment with.

The Community. The LLVM project now includes far more then LLVM: it is an entire compiler tool chain with a C/C++ compiler, assembly tools, a linker, a debugger, and static analysis tools. The work of the community on these projects and also the work of third party compilers, such as GHC, benefit all compilers based on LLVM and improve tool support.

3. How GHC works

Before discussing LLVM and how it fits into GHC's compilation process in Sections 4 and 5, this section details the aspects of GHC's design that are relevant to the LLVM backend.

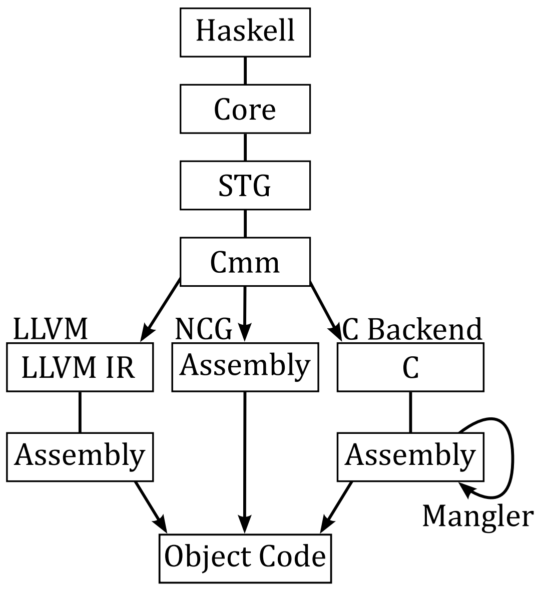

Figure 1. The GHC pipeline

3.1 The GHC pipeline

Figure 1 outlines GHC's compilation pipeline. The three main intermediate languages in that pipeline are:

Core. GHC's main optimisation engine is implemented in the form of a large number of program analysis and source-to-source transformation steps on the intermediate language Core. Core is a typed lambda calculus —specifically, it is an instance of System $F_C(X)$ [26]— and as such far removed from the process of target code generation. Hence, it plays no role in the design of the LLVM backend. Nevertheless, it is central to one of the areas of major innovation in GHC, which highlights our previous point that target code generation is essentially an unwelcome distraction to most GHC developers.

STG-language. This is an A-normalised [12] lambda calculus, which serves as the language of the *Spineless Tagless G-Machine (STGM)* [23] – the abstract machine that defines GHC's execution model. This execution model was originally the basis of the C backend, and hence, strongly impacts a number of the design choices in the target code generation. We will discuss the STG-machine and its impact on code generation in more detail in the following subsection.

Cmm. Cmm is a variant of the C-- language [24], which in turn could be described as a subset of C with extensions to facilitate the use as a low-level intermediate language — for example, it supports tail calls and the integration with a garbage collector. As most of the more complex features of C--, such as its runtime system, aren't supported in Cmm, it is even closer to being a subset of C. In fact, before GHC included Cmm, it used an intermediate language called *Abstract C* instead. Cmm is the starting point for the two original code generators, the C backend and the NCG, and it also serves as the input to our LLVM backend. It is central to developing a GHC backend and we will discuss it in detail in Subsection 3.4.

The dependence of GHC's code generators on Cmm is obvious in Figure 1, where the pipeline splits after Cmm depending on which backend is used. The NCG generates assembly directly from Cmm, whereas the C backend and the new LLVM backend generate C and LLVM IR, respectively. The generation of assembly from C and LLVM IR is then left to supporting tools, namely the GNU C compiler and the LLVM tools, respectively. The C backend additionally runs a post-processing tool, a Perl script with target code-specific regular expressions, over the assembly to further optimise it.

We will discuss the exact reasons for starting from Cmm in the LLVM backend in detail in Section 5. Before we can do this, we first need to introduce some of the design of the STG-machine and the Cmm intermediate language.

3.2 Spineless Tagless G-Machine

The STG-machine essentially comprises three parts: (1) the STG-language from Figure 1, (2) an abstract machine configuration consisting of a register set, heap, stack, etc., and (3) an operational semantics that defines in which way the various constructs of the STG-language alter the machine configuration upon execution. The first component, the STG-language itself, is not important for the LLVM backend as we translate the lower-level Cmm to LLVM IR.

However, the remaining two components, the machine configuration as well as the operational semantics are crucial to understanding the LLVM backend as it is the ultimate purpose of the backend to map these two components onto the target machine architecture — or more precisely, to map it onto the LLVM machine configuration and LLVM IR code, respectively. In theory, it is LLVM's responsibility to map STG-configurations and programs encoded in LLVM IR to the various concrete architectures. In practice, the design of the LLVM backend requires us to understand how LLVM IR maps to concrete architectures to generate efficient code.

In particular, we need to represent the heap, stack, and machine registers of the STG-machine on top of LLVM. As Cmm is specialised to GHC and the translation of STG-language programs, the Cmm code follows certain idioms and includes specific language constructs to handle the components of the STG machine configuration. Of particular importance is the register set of the STG-machine, which we will call *STG registers* in the following.

3.3 STG Registers

Abstract machines usually define the most frequently accessed components of their machine state as abstract machine registers to suggest that these are mapped to hardware registers for optimal performance. In the case of GHC, these abstract machine registers are also central for the interaction with the runtime system (RTS), which is written in C, Cmm, and some snippets of assembly. The STG registers function as an interface between generated code and the runtime system. In other words, the mapping of STG registers to the hardware registers and memory locations of the target architecture are *hard-wired* into the runtime system. The STG Registers include a stack and heap pointer, as well as a set of general registers that are used for argument passing.

Currently, there are two different ways in which GHC implements STG registers; they are called *unregisterised* and *registerised* mode, respectively. Unregisterised mode is the simple approach where all STG registers are stored in memory on the heap. Due to the frequent use of STG registers in GHC-generated code, this simple approach comes with a significant performance penalty and is mainly meant for porting and bootstrapping GHC on a new architecture. In unregisterised mode, GHC's C backend generates standard C code and omits the post-processing phase indicated in Figure 1.

In contrast, registerised mode uses the hardware registers of the target architectures to store at least the most important STG registers — this process if often referred to as *register pinning*. As there are far too many STG machine registers to store them all in real registers though, some still need to be stored in memory. This technique alone has a dramatic effect on the speed of programs.

As these registers are used by GHC's C-based runtime system, implementing the STG registers in a different manner then either of the two currently supported would involve significant changes to the RTS, increasing the development and maintenance effort. Hence an appropriate mapping of the STG registers can be a considerable challenge for a backend since it requires explicit control over register allocation, something not offered by many compiler targets, including LLVM.

3.4 Cmm

As depicted in Figure 1, Cmm is the final backend-independent intermediate representation used by GHC, and serves as a common starting point for the backend code generators. Cmm is based on the C-- language, but with numerous additions and omissions. The most important difference is that Cmm doesn't support any of C--'s runtime interface features. In C--, these features provide support for implementing accurate garbage collection and exception handling. Instead of involving Cmm, GHC uses a portable garbage collector, implemented in the runtime system, that requires no explicit support from the backends, but depends on the idiomatic generation of Cmm code by GHC.

Overall, Cmm is designed to be a minimal procedural language. It supports just the features needed to efficiently abstract away hardware and little more. The prominent features of the language are:

1. Unlimited variables, abstracting real hardware registers;
2. Simple type system of either bit types or float types;
3. Powerful label type and sections which can be used to implement higher-level data types;
4. Functions and function calling with efficient tail call support. Functions don't support arguments though, the STG registers are used instead to explicitly implement the calling convention used;
5. Explicit control flow with functions being comprised of blocks and branch statements;
6. Direct memory access;
7. A set of global variables that represent the STG registers; and
8. Code and data order in Cmm is preserved in the compiled code. GHC uses this property for implementing one particular optimisation, which we will examine in detail in the next subsection.

Cmm greatly simplifies the task of a backend code generator as the non-strict and functional aspects of Haskell have already been handled and the code generators instead only need to deal with a fairly simple procedural language. Figure 2 displays an example of Cmm. It demonstrates a large portion of the Cmm language, such as its types, variables, control structures and use of code and data labels.

3.5 Cmm data & code layout

One of the requirements Cmm places on a backend is that the generated object code has the same order of the data and code sections as the Cmm code has. If a data and code section are adjacent in the Cmm code they are expected to be adjacent in the final object code. This is a problematic requirement as C compilers and other tools take the liberty to reorder code and data sections. Hence, this requirement accounts for much of the magic performed by the Perl script realising the assembly post-processing for the GHC's C backend. It turns out to be a problem for the LLVM backend, too.

```
section "data" {
    fibmax:
        bits32 35;
}

fib()
{
    bits32 count; count = R1;
    bits32 n2; n2 = 0;
    bits32 n1; n1 = 1;
    bits32 n; n = 0;

    if (count == 1 || bits32[fibmax] < count) {
        n = 1;
        goto end;
    }

 for:
    if (count > 1) {
        count = count - 1;
        n = n2 + n1;
        n2 = n1;
        n1 = n;
        goto for;
    }

 end:
    R1 = n;
    jump StgReturn;
}
```

Figure 2. Cmm example: Fibonacci numbers

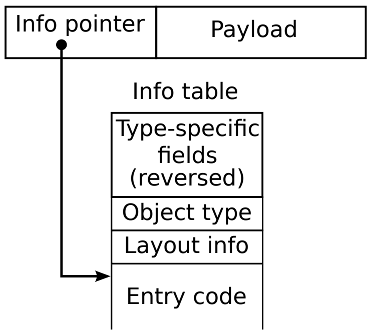

Figure 3. GHC's optimised TNTC heap layout

But before getting into the details of the LLVM backend, let us review the reason for the onerous constraint of the Cmm intermediate language. GHC uses it to implement an optimisation known as tables-next-to-code (TNTC). The basic idea is to lay the metadata of a closure right before the code for the closure itself. The metadata, which we call an *info-table*, is required by the runtime system for each closure. With that layout, both the closure's evaluation code and its metadata can be accessed from a single pointer.

A graphical representation of this layout is in Figure 3. The first word of a closure is its *info pointer* that refers to the first instruction of the closure's *entry code*. The remaining fields of the closure, its *payload*, contains a function's free variables or a data constructor's arguments.

A closure's entry code is executed when the closure is evaluated — i.e., when a lazily evaluated piece of code, a *thunk*, is forced or

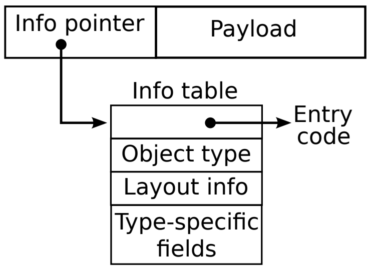

Figure 4. GHC Unoptimised Heap Layout

when a function closure is entered to apply it to some arguments. The entry code of closures representing data structures that are in normal form returns a value identifying the corresponding data constructor or similar. By indexing backwards from a closure's info pointer, the runtime system can access the info-table that contains layout information to assist garbage collection and other metadata.

Without the TNTC optimisation, the first word of a closure does not refer directly to the entry code, but instead to the info-table, as depicted in Figure 4. The info-table, in turn, explicitly stores a pointer to the entry code in an additional field. Hence, without the TNTC optimisation, info tables use one more word of memory and, more importantly, executing a closure's entry code, when it is evaluated, requires two pointer lookups instead of one. This is costly as Haskell code creates and evaluates closures at a rapid rate.

In summary, due to the frequent closure entry of Haskell code, the GHC designers chose to bake a layout constraint into Cmm that is hard to meet with conventional backend technology, such as compiling via C or using a general-purpose framework, such as LLVM. We will look into the capabilities of LLVM in more detail in the following section.

4. How LLVM works

The *Low Level Virtual Machine (LLVM)* is an open source, mature optimising compiler framework whose development started in 2000 as part of Lattner's Masters thesis [18]. Today, it provides a high-performance static compiler backend, but can also be used to build just-in-time compilers and to provide mid-level analyses and optimisation in a compiler pipeline. Its main innovation is in the area of life-long program analysis and optimisation [19] — i.e., it supports program analysis and optimisation at compile time, link time, and runtime. Our GHC LLVM backend currently ignores the link-time and runtime analysis and optimisation capabilities and uses LLVM as a conventional static backend. Hence, we are most concerned with LLVM's abstract machine language that serves as input the LLVM pipeline.

4.1 The LLVM assembly language

The LLVM assembly language, *LLVM IR*, is the input language which LLVM accepts for code generation. However, it also acts as LLVM's internal intermediate representation for program analysis and optimisation passes. The IR has three equivalent representations: a textual representation (the assembly form), an in-memory representation, and a binary representation. The textual representation is useful in a compiler pipeline where individual tools communicate via files, as well as for human inspection. The in-memory representation is used internally, but also whenever a compiler links to LLVM as a library to avoid the overhead of file input and output. The binary representation is used for compact storage — it occupies less storage than the textual format and can be read more efficiently.

```
define i32 @pow( i32 %M, i32 %N ) {
LoopHeader :
    br label %Loop
Loop :
    %res = phi i32 [1, %LoopHeader], [%res2, %Loop]
    %i = phi i32 [0, %LoopHeader], [%i2, %Loop]
    %res2 = mul i32 %res , %M
    %i2 = add i32 %i, 1
    %cond = icmp ne i32 %i2 , %N
    br i1 %cond , label %Loop , label %Exit
Exit :
    ret i32 %res2
}
```

Figure 5. LLVM code to raise an integer to a power

The LLVM IR is low-level and assembly-like, but it maintains higher-level static information in the form of type and dataflow information — the latter due to using *static single assignment (SSA) form* [9]. SSA form guarantees that every variable is only assigned once (and never updated), and hence, strongly related to functional programming [6]. The design goal in combining a low-level language with high-level static information is to retain sufficient static information to enable aggressive optimisation, while still being low-level enough to efficiently support a wide variety of programming languages.

The main features of LLVM's assembly language are:

1. Unlimited virtual registers, abstracting real hardware registers;
2. Low-level assembly with higher-level type information;
3. Static single assignment form (SSA) with phi (ϕ) function;
4. Functions and function calling with efficient tail call support;
5. Explicit control flow with functions comprising blocks and branch statements; and
6. Direct memory access, as well as a type-safe address calculation instruction, `getelementptr` facilitating optimisations.

The single-assignment property of the SSA form requires the use of phi (ϕ) functions in the presence of low-level control flow with explicit branches. A phi function selects the value to be assigned to a virtual register in dependence on the edge of the control-flow graph along which execution reached the phi function. SSA form is well-established as a type of intermediate representation that simplifies the implementation of code analysis and optimisation.

The above feature list of the LLVM IR has much in common with the feature list of Cmm (in Section 3.4). We will compare the two in detail in Section 5.1, where we discuss the translation. For the moment, let's have a look at a concrete piece of LLVM IR code.

The code in Figure 5 contains one complete LLVM function, which is made up of a list of *basic blocks*, each preceded 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`). Variable names preceded by a percent symbol, such as `%res` and `%i`, denote virtual registers. Virtual registers are introduced by the unique assignment that defines them — just as in a let-binding. All operations are annotated with type information, such as `i32`, which implies an integer type of 32 bits. Finally, the `Loop` block starts with two phi functions. The first one assigns to `%res` either the constant `1` or the value stored in register `%res2` depending on whether execution entered the `Loop` block from the block `LoopHeader` or from `Loop` itself.

All LLVM code is defined as part of an *LLVM module*, with modules serving as compilation units. 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. Global variables are as expected, and are prefixed with the @ symbol, as are functions, to indicate that they are actually pointers to the data and have global scope. This also distinguishes them from local variables which are prefixed with the % symbol.

4.2 Comparing C-- and LLVM

As mentioned previously, GHC's Cmm language is based on the C-- language. The goals of the C-- project were not unlike those of the LLVM project. There is, however, an important difference between the two: LLVM is geared towards supporting aggressive optimisation of a universal language and C-- towards supporting high-level language features such as garbage collection with no overhead. It is interesting though that despite these differences both projects independently developed very similar features. This might suggest that a universal low-level language needs to support a certain essential set of features. It is also interesting to see that over its lifetime, LLVM's design has in some areas moved towards that of C--. A few examples of features that C-- supported in its initial design and that LLVM only added later are:

- LLVM's type system originally was similar to C, supporting signed and unsigned variations of `char, byte, int` and `long`. Now its type system is much closer to C--, having simply `bitsN` types and not distinguishing between signed or unsigned. LLVM also used to exclusively use overloaded operations, such as addition and division, but now increasingly has separate instructions for the different types.
- LLVM at first did not support declaring the calling convention of functions and calls, they have only been added later.
- LLVM originally contained a `malloc` and `free` instruction but these have very recently been removed.
- LLVM now has direct support for implementing garbage collection. This is not as complex or as versatile as the compile and runtime interface supported by C--, but it works in a similar manner: frontend compilers annotate their code with safe points and call special functions in their LLVM code that trigger a compiler plug-in, which they need to supply, to add the information needed by their garbage collector to the code. Our backend doesn't use this support, though as the garbage collector implemented by GHC doesn't require it.

5. LLVM backend design

As shown in Figure 1, our LLVM backend uses Cmm as its input language, just like the other two backends. In principle, we could have used STG-language as our input language, to try and use the higher-level information in the STG-language to generate better code. However, that would have meant duplicating much of the existing functionality in the STG-to-Cmm phase, which not only deals with sophisticated language features, such as laziness and partial application, but also runtime system considerations, such as the generation of metadata for the garbage collector. Instead of replicating this functionality, it seems more economical to fix any shortcomings in the Cmm code generator and in the Cmm language if and when we identify any situation where the LLVM backend doesn't have the information it needs. This hasn't happened so far. Moreover, sharing as much code as possible between the backends simplifies maintaining ABI compatibility between the new LLVM backend and the existing backends, which is important to enable linking to modules and libraries compiled with other backends. Finally, there is ongoing work in GHC to move to a new Cmm code generator and a slightly changed Cmm representation [25]. Once complete, this work should improve the code generated by all backends, making it complementary to the LLVM backend instead of competitive.

Despite all backends compiling off Cmm, the design and implementation of a translation phase to LLVM IR raises a number of conceptual problems that are unique to the LLVM backend: (1) the mapping of Cmm language constructs to LLVM IR, (2) the generation of LLVM's SSA form and especially of the phi functions, (3) an efficient implementation of the STG registers, and (4) the implementation of Cmm's strict code and data layout constraints. We will address these issues individually in the following subsections.

5.1 Compiling Cmm to LLVM IR

The Cmm and LLVM IR were designed with a similar goal in mind: to be a minimal language to abstract hardware. The primary difference is LLVM's broader focus, aiming to support multiple languages and aggressive optimisation of the code, whereas Cmm, as used in GHC, is skewed towards compiling Haskell-like languages.

To support high-level data types, Cmm uses a label system that works like assembly labels for implementing data types. There is no type information, and arrays and record structures are implemented in the same manner. LLVM instead supports such high-level data types, such as arrays and structures explicitly. Nevertheless, translating between the two is fairly straightforward, especially since many of the harder cases, such as a Cmm data structure with labels at non-start positions, aren't used by GHC and so don't need to be supported — these features were inherited from C--.

Another minor difference is that LLVM's preferred way of accessing memory is a special instruction, `getelementptr`, that takes a pointer type, such as an array, and an index, returning a type-safe pointer. In contrast, Cmm uses explicit pointer arithmetic. LLVM supports this, too, but it prevents some worthwhile code optimisations. Initially we simply used pointer arithmetic in the LLVM backend but have recently changed to using the `getelementptr` instruction, primarily as part of some work to give LLVM better aliasing information.

Many other aspects of Cmm and LLVM IR are rather similar and instead of discussing all features in detail, Table 1 provides a summary of the relationship. As a concrete example, consider the translation of the Cmm code of Figure 2 into unoptimised LLVM code in Figure 6. After improving the code with LLVM's optimiser, the code is more compact as shown in Figure 7. By relying on the LLVM optimiser, instead of trying to generate better-optimised LLVM code directly, we could keep the LLVM backend simpler — after all, we want to offload as much work as possible onto LLVM.

5.2 Dealing with LLVM SSA form

As we discussed in Section 4, LLVM code must be in SSA form — i.e., all LLVM virtual registers are immutable, single-assignment variables. In contrast, all of Cmm's variables are mutable; so, we need to handle the conversion to SSA form as part of the LLVM backend. The conversion of arbitrary code into SSA form is a well understood problem; however, it requires a fair amount of implementation work. Thankfully, LLVM provides us with an alternative option that is far simpler: instead of LLVM's virtual registers, we can use stack allocated variables.

We translate each mutable Cmm variable into an LLVM variable allocated on the stack using the `alloca` instruction. This instruction returns a pointer into the stack that can be read from and written to just like any other memory location in LLVM by using

Cmm	LLVM
Basic Types	
Fixed set of integer and floating point types: *i8, i16, 132, 164, i128* *f32, f64, f80, f128*	Support for any size N bit type and a fixed set of floating point types: *i1, i2, i3... i32, ... iN* *float, double, x86-fp80, fp128*
High Level Types	
Supports a *label* type that represent the address of the location it's declared at. This can be used to implement higher level types such as arrays and structures: *Array*: cmmLabel {i32, i32, i32, i32} *Structure*: cmmLabel {i32, f32, f64}	Has explicit support for high level types such as arrays and structures: *Array*: [4 x i32] *Structure*: {i32, float, double}
Variables	
Unlimited typed local variables. Global data is represented through untyped labels, all load and store operations are instead typed.	Unlimited typed local and global variables, however LLVM's use of SSA form means Cmm local variables don't map directly to LLVM local variables. Stack allocated variables are used instead.
Code Module Structure	
A module consists of global variables and functions. Functions don't support arguments or return values, the STG registers are used for this purpose instead. Functions consist of a list of blocks. All control flow between blocks is explicit.	A module consists of global variables, functions, type aliases, metadata and a data layout specification. Functions support arguments and a single return value.
Expressions	
Literals, memory reads, machine operations and STG register reads	*Literals, memory reads and machine operations.* LLVM has a full coverage of the machine operations that Cmm supports and not much more, the mapping is nearly 1 to 1.
Statements	
Comments, assignment, memory writes, unconditional branch, conditional branch, multi-way branch, tail calls and function calls. Cmm also supports calls to a group of functions called *CallishMachOp*. These are maths functions such as sin and log. On hardware which supports them they should become CPU instructions, otherwise they are turned into calls to the C standard library.	LLVM directly supports all of the statements that Cmm supports and a few more. Interestingly LLVM also supports many of the Cmm *CallishMachOp* instructions and in a similar manner where in LLVM they are termed intrinsic functions.

Table 1. Mapping of Cmm and LLVM languages

```
%fibmax_struct = type {i32}
@fibmax = global %fibmax_struct {i32 35}

define cc 10 void @fib( i32 %Base_Arg,
      i32 %Sp_Arg, i32 %Hp_Arg,i32 %R1_Arg) {
cP:
    [...]
    %R1_Var = alloca i32, i32 1
    store i32 %R1_Arg, i32* %R1_Var
    %cf = alloca i32, i32 1
    %cl = alloca i32, i32 1
    %ck = alloca i32, i32 1
    %cj = alloca i32, i32 1
    %nQ = load i32* %R1_Var
    store i32 %nQ, i32* %cf
    store i32 0, i32* %cl
    store i32 1, i32* %ck
    store i32 0, i32* %cj
    %nR = load i32* %cf
    %nS = icmp eq i32 %nR, 1
    br i1 %nS, label %cT, label %nU
nU:
    %nV = bitcast %fibmax_struct* @fibmax to i32*
    %nX = load i32* %nV
    %nY = load i32* %cf
    %nZ = icmp ult i32 %nX, %nY
    br i1 %nZ, label %cT, label %n10
n10:
    [...]
```

Figure 6. Partial output of LLVM backend with Figure 2 as input

```
%fibmax_struct = type { i32 }
@fibmax = global %fibmax_struct { i32 35 }

define cc 10 void @fib( i32 %Base_Arg,
      i32 %Sp_Arg, i32 %Hp_Arg,i32 %R1_Arg) {
cP:
  %nS = icmp eq i32 %R1_Arg, 1
  br i1 %nS, label %c12, label %nU
nU:
  %nX = load i32* getelementptr inbounds
          (%fibmax_struct* @fibmax, i32 0, i32 0)
  %nZ = icmp ult i32 %nX, %R1_Arg
  br i1 %nZ, label %c12, label %c13.preheader
c13.preheader:
    [...]
}
```

Figure 7. Output of LLVM Optimiser with Figure 6 as input

explicit `load` and `store` instructions. Code using stack-allocated variables instead of virtual registers is generally slower, but LLVM includes an optimisation pass called *mem2reg,* which is designed to correct this. This pass turns explicit stack allocation into the use of virtual registers in a manner that is compatible with the SSA restriction by using phi functions. In effect, *mem2reg* implements the SSA conversion for us.

5.3 Handling the STG registers

The efficient treatment of the STG registers was one of the major challenges we faced in writing the LLVM backend. While the LLVM backend could easily implement the STG registers using *un-*

registerised mode, where they are all stored in memory, this would lead to poor performance. Hence, we need to support *registerised mode* to map as many of the STG registers as possible to hardware registers, and we want to do that such that it yields the same register mapping as used by the other two backends. This is crucial for ABI compatibility, as discussed.

Register mapping is a straight forward affair for the NCG as it has full control over register allocation. GHC's NCG register allocator is aware of the special status of the STG registers and simply reserves the appropriate hardware registers for exclusive use as STG registers — i.e., it aliases the STG register with some hardware registers. The situation is similar for the C backend. Although, ANSI C does not offer control over hardware-specific registers, GNU C provides an extension, called *global register variables*, which facilitates the same approach of reserving fixed hardware registers for specific STG registers throughout the code.

LLVM does not provide this option. Instead, our solution is to implement a new calling convention for LLVM that passes the first n arguments of a function call in specific hardware registers. We choose the hardware registers that we would like to associate with STG registers. Then, the LLVM backend compiles each Cmm function such that the corresponding LLVM function uses the new calling convention with the appropriate number of parameters. Furthermore, the generated LLVM code passes the correct STG registers as the first n arguments to that call.

As a consequence, the values of the STG registers are in the appropriate hardware registers *on entry to any function*. This is in contrast to the other two backends, where the STG registers are also pinned to their hardware registers throughout the body of a function. However, to guarantee the correct register assignment of function entry it is sufficient to ensure that the runtime system finds the registers in the correct place and that LLVM code can call and be called from code generated by other backends. In fact, it is an improvement over the strategy of the other two backends, as the n hardware registers can temporarily be used for other purposes in function bodies if LLVM's register allocator decides it is worthwhile spilling them. In most cases though, simply leaving the STG registers in the hardware registers is the best allocation and LLVM is capable of recognising this.

The only down side of a new calling convention is that its addition requires modifying the LLVM source code. However, our extension has recently been accepted upstream by the LLVM developers. It is now included in public versions of LLVM since version 2.7, which was released in May of 2010 — i.e., as long as version 2.7 or later is being used, GHC works with a standard LLVM installation.

5.4 Handling Cmm Data and Code layout

As discussed in Section 3.5, Cmm's requirement to preserve the ordering of data and code layout is uncommon and causes problems with backends other than the NCG (which generates the assembly sections explicitly). Unfortunately, GHC uses this property of Cmm to implement the tables-next-to-code optimisation, which is fairly significant with an about 5% reduction in runtimes.

As the C backend can't meet the layout requirement with either ANSI C or through one of the GNU C extensions, it resorts to an extra pass over the assembly code produced by the GNU C compiler to rearrange assembly sections and rewrite the code.

The LLVM backend faces the same problem as the C backend, as there is no explicit support for ordering code sections in LLVM. Fortunately, we found a technique to realise the ordering constraint by using the *sub-sections* feature of the GNU Assembler (*gas*). This feature facilitates the specification of a *numbered subsection* whenever assembly is placed into a particular assembly section. When gas compiles the assembly to object code, it combines subsections

```
.text 12
sJ8_info:
  movl  $base_SystemziIO_hPrint2_closure, (%ebp)
  movl  $base_GHCHandleziFD_stdout_closure, -4(%ebp)
  addl  $-4, %ebp
  jmp   base_GHCziIOziHandleziText_hPutChar1_info

.text 11
sJ8_info_itable:
  .long  Main_main1_srt-sJ8_info
  .long  0
  .long  327712
```

Figure 8. GNU Assembler Subsections to implement TNTC

in ascending numerical order and creates only one section including all the numbered subsections. In other words, the subsections are purely a structure that exists in the assembly, but does not appear explicitly in the object code. To guarantee that a closure's metadata appears immediately before the closure's entry code, we simply place the metadata in section '`text` n' and the entry code in section '`text` $\langle n+1 \rangle$', making sure that no other code or functions use those subsections. This is illustrated at an example in Figure 8.

While this approach works well it does create a portability problem as it only works with the GNU Assembler. Fortunately this covers two of the three major platforms GHC is supported on, Linux and Windows. On Mac OS X though we are unable to use this technique and so for now have resorted to post processing the assembly produced by LLVM in a manner similar to the C backend. While this is regrettable it is important to note that the mangler[2] used by the LLVM backend consists of only about 180 lines of Haskell code, of which half is documentation. The C mangler by comparison is around 2000 lines of Perl code as it has to handle multiple platforms and far more than simple assembly code rearrangement. We are planning, for the future, to move to a purely LLVM-based solution by extending LLVM to explicitly support associating a global variable with a function. This approach might enable better code optimisation; for example, by performing global constant propagation with the info table values.

6. Evaluation of the LLVM backend

Next, we evaluate the new LLVM backend in comparison to GHC's existing C backend and NCG. The evaluation is in two parts: first, we consider the complexity of the backends themselves, and second, we analyse the performance of the generated code. The backend complexity is a primary concern for GHC developers, whereas code performance concerns both developers and users of GHC.

6.1 Complexity of backend implementations

As a simple metric for the code complexity of the three backends, we compare their respective code size. This gives us an indication of the amount of work initially required to implement them as well as the effort that is spend on maintenance. Table 2 displays the code size of the various components of the three backends.

The LLVM backend is the smallest at 3,133 lines of code. The C backend is over 70% larger at 5,382 lines. The NCG is by far the largest, being 6 times larger than the LLVM backend and 4 times larger than the C backend — it totals 20,570 lines.

In addition to plain code size, we also need to consider the structural and conceptual complexity, particularly for the C backend which doesn't seem an unreasonable size. The C backend con-

[2] We are tentatively calling this pass, the Righteous Mangler, in line with the established naming convention

Lines of code of the GHC backends

C	**Total**	**5382**
	Compiler	1122
	Includes	2201
	Assembly Processor	2059
NCG	**Total**	**20570**
	Shared	7777
	X86	5208
	SPARC	4243
	PowerPC	3342
LLVM	**Total**	**3133**
	Compiler	1865
	LLVM Module	1268

Table 2. GHC backend code sizes

sists of three distinct components: (1) the actual compiler that maps Cmm to C code; (2) the C header files included in the generated C code; and (3) the evil mangler, a Perl script post-processing the generated assembly. The C headers define a large number of macros and data structures that decrease the work required by the code generator and also deal with platform specific issues, such as word size. The C headers are fairly sophisticated, but the arguably most complex part of the C backend is the evil mangler, which is implements the TNTC optimisation as well as a variety of other optimisations, such as removing each C function prologues and epilogues. The fragile Perl code uses a large number of regular expressions that need to be updated regularly for new versions of GCC.

The NCG consists of a shared component plus a platform-specific component for each supported architecture. The design is fairly typical of a NCG. The shared component consists a general framework for abstracting and driving the pipeline as well as a register allocator. Each platform specific component is responsible for the rest of the pipeline, principally consisting of instruction selection and pretty printing of the assembler code.

The LLVM backend comprises two components: (1) the compiler itself and (2) a code module for interfacing with LLVM. It has none of the complexity of the C backend with its sophisticated assembly post-processing or of the NCG with its size and architecture-specific code. It's also nearly platform independent; it needs to know the word size, endianness and the mapping of STG registers to hardware registers. All of this information is already used elsewhere in GHC and so isn't specific to the LLVM backend.

6.2 Performance

To compare the quality of the generated code, we will consider the runtime, but also other metrics, such as compilation times and the size of the compiled code. We used a Core 2 Duo 2.2GHz machine running a 32 bit Linux OS and set the runtimes of the LLVM backend to be the baseline, except where otherwise specified. So positive percentages for either the C backend or NCG mean that they are slower than the LLVM backend by that percentage and negative percentages mean they are faster by that percentage. First, we investigate the NoFib benchmark suite and then some interesting individual examples. Also, as the NCG is GHC's default backend and as the GHC developers are looking at deprecating the C backend, we will focus on comparing against the NCG.

NoFib [22] is the standard benchmark suite for GHC. It is developed alongside GHC and used by developers to test the performance impact of changes to GHC. In Table 3, we see the runtimes of the NCG and C backend against the LLVM backend.

There is little difference between the three backends. The NCG comes out with the best overall runtime, ahead of the LLVM backend by 0.1%. The C backend comes in last, trailing 2.7% behind the LLVM backend. The tables only includes individual benchmarks where the runtimes vary significantly between backends. We investigated each of these benchmarks individually to determine the cause of the difference. We didn't find any cases where the C backend had any conceptual advantage. Where the C backend performed poorly, this was a combination of the at times awkward mapping of Haskell to C and performance bugs with GCC.

NCG and C backend against LLVM backend ($\Delta\%$)

Program	NCG Runtime	C Runtime
atom	-6.2	-0.8
comp_lab_zift	-4.6	-0.9
cryptarithm1	-3.4	0.9
hidden	4.7	10.9
integer	-1.0	-1.3
integrate	2.8	8.3
simple	1.5	16.6
transform	4.0	5.7
treejoin	-2.8	4.6
wave4main	6.8	12.4
wheel-sieve2	-3.4	-2.8
(79 more)	..	..
-1 s.d.	-3.1	-2.0
+1 s.d.	3.0	7.6
Average	-0.1	2.7

Table 3. NoFib runtimes of all three backends

```
Main_runExperiment_entry()
  c1GU:
    Hp = Hp + 36;
    if (Hp > HpLim) goto c1GX;
    I32[Hp - 32] = s1Gh_info;
    I32[Hp - 24] = I32[Sp + 12];
    I32[Hp - 20] = I32[Sp + 0];
    I32[Hp - 16] = I32[Sp + 4];
    I32[Hp - 12] = I32[Sp + 8];
    I32[Hp - 8] = ghczmprim_GHCziTypes_ZC_con_info;
    I32[Hp - 4] = I32[Sp + 12];
    I32[Hp + 0] = Hp - 32;
    R1 = Hp - 6;
    Sp = Sp + 16;
    jump (I32[Sp + 0]) ();
  c1GY:
    R1 = Main_runExperiment_closure;
    jump stg_gc_fun ();
  c1GX:
    HpAlloc = 36;
    goto c1GY;
```

Figure 9. A typical Cmm function produced by GHC

Comparing the NCG against the LLVM backend, further testing showed that the performance difference was the greatest for *atom*, *hidden* and *wave4main*. However, in all three cases, no particular feature of the code generation seemed to be responsible. It was just a matter of a better default instruction selection.

All this raises an important question: why do we get such similar results with very different code generators? Especially, if we consider that GHC's NCG optimisation pass consists of just branch-chain elimination and constant folding, whereas LLVM implements around 60 different optimisation passes. We conjecture that the reason for the similar performance is the Cmm code they all use as in-

LLVM optimiser against O0 (Δ%)

	O1	O2	O3
-1 s.d.	-4.9	-6.7	-6.3
+1 s.d.	3.0	2.1	4.0
Average	-1.0	-2.4	-1.3

Table 4. NoFib runtimes of LLVM at different optimisation levels.

	mmult		laplace		fft	
#cores	1	8	1	8	1	7
NCG	13.38s	1.68s	4.75s	1.44s	8.88s	2.06s
LLVM	4.64s	0.62s	2.98s	1.15s	8.75s	2.02s
Speed up	2.88	2.71	1.59	1.25	1.01	1.02

Table 5. LLVM versus NCG performance for Repa benchmarks

```
import qualified Data.Vector.Unboxed as U
main = print $
          U.sum $ U.zipWith3 (\x y z -> x * y * z)
                (U.enumFromTo 1 (100000000::Int))
                (U.enumFromTo 2 (100000001::Int))
                (U.enumFromTo 7 (100000008::Int))
```

Figure 10. Vector Zip3 benchmark

```
collatzLen :: Int -> Word32 -> Int
collatzLen c 1 = c
collatzLen c n = collatzLen (c+1) $
    if n `mod` 2 == 0 then n `div` 2 else 3*n+1

pmax x n = x `max` (collatzLen 1 n, n)

main = print $ foldl pmax (1,1) [2..1000000]
```

Figure 11. Hailstone benchmark

put: it just isn't easily optimised. Much of the Cmm code that GHC produces is essentially memory bound, a side effect of Haskell being a lazy evaluated language and so there is often very little register pressure or choice in the instruction selection, which is why the NCG is able to perform close to LLVM. Figure 9 contains some typical Cmm code, illustrating the problem.

To further test our conjecture, we ran the NoFib benchmarks with the optimisation level of LLVM set to the various supported default groups: *-O0, -O1, -O2* and *-O3*. The results are in Table 4.

NoFib however doesn't tell us the full story concerning performance, specifically due to the idiomatic Cmm of GHC. This becomes, for example, apparent in code using stream fusion [8] and highly optimised array code, such as that of the parallel array library Repa [17]. The Cmm code of the compute-intensive, tight inner loops of these libraries generally suffer from high register pressure and can benefit from smart instruction ordering, which leaves considerable scope for LLVM to optimise performance. The considerable impact that LLVM's optimisations can have on such code is quantified in Table 5, where we compare the single-threaded and multi-threaded performance of NCG and LLVM-generated code for three Repa benchmarks (see [17] for details on these benchmarks).

We further investigated two simple benchmarks featuring tight loops: (1) *zip3*, Figure 10, uses the high-performance `vector` library, based on array fusion; and (2) *hailstone*, Figure 11, relies on list-fusion from the standard Prelude and unboxed integers. We evaluated both benchmarks using the Criterion benchmarking library [21] and looking at the resulting kernel density estimates.

Figure 12 shows the kernel density estimates for both benchmarks using the three backends. For zip3, the LLVM backend comes out clearly in front with a mean runtime of 334ms, the C backend second with 423ms and the NCG last with a mean of 590ms. The generated Cmm code consists of 3 functions that produce the three enumerated lists. Each calls a common comparator function that checks whether the end of the list has been reached. The LLVM backend aggressively inlines the comparator function, saving a jump instruction for each of the three list enumerations. The C backend generates remarkably similar code to the NCG, the difference simply seems to be in the ordering of some branches and basic blocks, with the C backend choosing the correct hot path.

For hailstone, we see that the C backend comes out in front with a mean of 567ms, the LLVM backend second with a mean of 637ms and the NCG last with a disappointing mean of 2.268s. The LLVM and C backends perform well for two reasons: (1) they both perform significantly better instruction selection and (2) they both inline a large amount of code. The C backend outperforms the LLVM backend due to slightly better branch ordering.

NCG and C backend against LLVM backend (Δ%)

Metric	NCG	C Backend
Object File Sizes	-12.8	-5.2
Compilation Times	-64.8	+35.6

Table 6. NoFib: Object file sizes and compile times

Table 6 lists the summary of the compile times and object file sizes for the NoFib suite. Both the NCG and C backend produce smaller code. This is currently a deficiency of LLVM: it does not yet optimise for code size. For compile times, the LLVM backend sits between the NCG and C backend. This is due to LLVM's additional optimisation passes, which incur an overhead compared to the NCG. We saw the considerable benefit of these optimisations in the runtimes of the Repa, zip3, and hailstone benchmarks.

6.3 LLVM's type system

An advantage of LLVM is its fairly strong type system and static checking of compiled code. While it doesn't approach the level of sophistication that Haskell programmers are used to, it does offer a system similar to C's. All variables and memory locations are typed, all operations obey strict type rules, and pointers are carefully distinguished from other types. For example to conduct pointer arithmetic, a pointer must first be cast to an integer of word width type for all arithmetic and then cast back to a pointer. We found this type system very helpful while implementing the LLVM backend, especially as there is usually little compiler support for such a low-level task as code generation.

To quantify the benefit of LLVM's checks, we scanned the source code revision history for the backend, looking at the bugs we still had to fix after the backend was able to compile a whole Haskell program. There were 15 fixes in total, after which the backend was capable of compiling GHC itself. Of these 15, 10 fixes were motivated by compile time errors generated by LLVM. Some of these were obvious bugs that would have also been discovered by a traditional assembler, but a few were more subtle. They generally related to pointer handling, such as one bug where we returned the pointer itself instead of the value it pointed to. For the 5 bugs that LLVM didn't pick up, two were related to generating incorrect function and data labels, one was an incorrectly compiled negate operation, one an incorrectly compiled label offset operation and one was due to a bug in the LLVM optimiser.

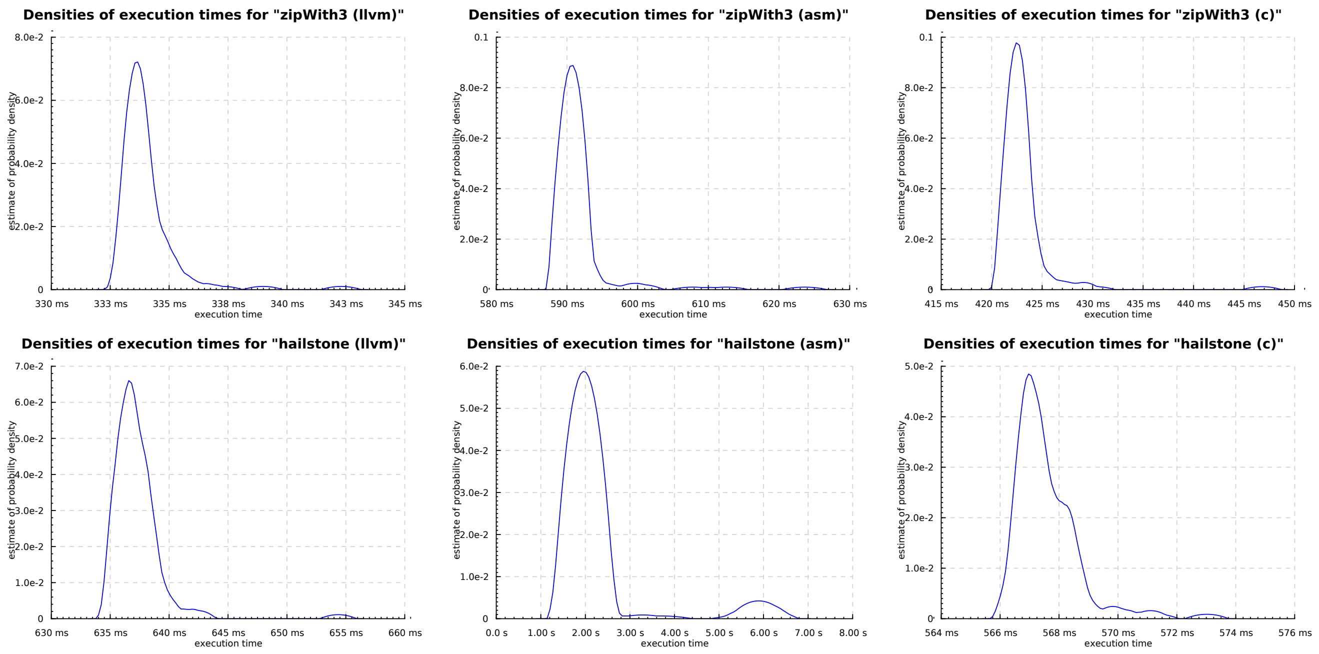

Figure 12. Runtimes of Hailstone and Zip3 benchmarks

The combination of LLVM's type system with the SSA form that requires every operation to be explicit is a significant help for compiler development. It's also rewarding to see type systems being used at such a low level. Although it was originally designed to enabled more aggressive optimisations, it does increase safety.

7. Related work

There is a large amount of work in the compiler community around LLVM, including static code generation, just-in-time code generators, and static analysis tools.

Schie recently added an LLVM backend to the Utrecht Haskell Compiler[3] (*UHC*), a research Haskell compiler, designed around the idea of implementing the compiler as a series of compilers [11, 29]. UHC's usual backend is a C code generator. His work produced some impressive results, which included a 10% reduction in the runtime of compiled programs. The UHC LLVM backend however didn't reach the stage of being able to handle the full Haskell language, instead working with a subset of Haskell. The results with the UHC backend can hardly be compared to our work, as GHC by default already generates code that is on average 40 times faster than that of UHC.

Other projects using LLVM include the following:

- *Clang*: A C, C++ and Objective-C compiler using LLVM as a backend target [1].
- *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 [4].
- *Pure*: A functional programming language based on term rewriting. Pure uses LLVM as a just-in-time compiler [13].
- *Unladen Swallow*: Google backed Python virtual machine with an LLVM based just-in-time compiler [5].
- *LDC*: A compiler for the D programming language using LLVM as a backend target [2].
- *llvm-lua*: A compiler for the Lua programming language using LLVM as a backend target [3].
- *SAFECode*: A compiler that takes LLVM bitcode as input and uses static analysis to produce a memory safe version of the same program [10].

Using LLVM isn't the only approach though to provide a relatively portable, easy to target, high performance compiler target, there are also high-level virtual machines, such as Microsoft's Common Language Runtime (*CLR*) [16], or the Java Virtual Machine (*JVM*) [20]. These are in some ways quite similar to LLVM, they all provide a virtual instruction set that abstracts away the underlying hardware and can be targeted by multiple programming languages. The functional programming language Clojure [15] for example targets the JVM with great success. Targeting the JVM or CLR also has the added benefits of direct access to the high quality libraries that come with both platforms. There are trade offs though, using a high level virtual machine means that many of your choices are made for you. Features such as garbage collection and exception handling are provided and as such you need to be able to efficiently map your language onto the design of these services, which may not always be possible, or at least efficient. Neither the CLR or JVM provide an option of compiling your code to native machine code, both use interpreters and just-in-time compilation for execution which generally leads to lower performance. LLVM doesn't include these high-level services and enables us to use infrastructure optimised for Haskell. It also permits us to choose between static compilation or interpretation with just-in-time compilation.

As well as the work around LLVM, there is also work being done in GHC around code generation. Ramsey et al. are redesigning the architecture of GHC's backend pipeline [25] to improve the code generated by GHC. A large part of this work is the design of a dataflow optimisation framework called Hoopl that can be used to easily compose together distinct passes. While there is some

[3] UHC was previously known as the Essential Haskell Compiler (EHC)

overlap in the optimisations that can be done with Hoopl and those implemented by LLVM, this work is mostly complementary to the LLVM backend as it is intended to replace the current STG to Cmm code generator with a much more modular design, not just duplicate optimisations present in LLVM. The end result of the work will be more efficient Cmm code passed to the LLVM code generator.

8. Conclusion

Our LLVM backend is clearly simpler, conceptually and in terms of lines of code, than the two previous backends. It effectively outsources a sophisticated part of GHC's compilation pipeline and frees developer resources to concentrate on issues that are more directly relevant to the Haskell community.

Our quantitative analysis shows that the LLVM backend, already in its current form, generates code that is on par with GHC's native code generator (the more efficient of the two current backends). For tight loops, as generated by the vector package, we even see a clear performance advantage of our backend.

The biggest disadvantage of the LLVM backend is currently its comparatively high compilation times with respect to the native code generator. This is partly to be expected as the LLVM backend is performing a lot more optimisation work on the code. We do expect to be able to improve on the compilation speed though as currently the LLVM backend interfaces with LLVM by using intermediate files and calling the LLVM command line tools, which wastes time parsing and pretty printing LLVM code. LLVM can also be used as a shared library, using entirely in-memory representations of the LLVM IR. By using this facility, we should be able to improve compilation speeds significantly.

Acknowledgements. We are grateful to Chris Lattner, Euguene Todler, Ben Lippmeier and Simon Marlow for technical advice during the development of the LLVM backend. We furthermore thank Ben Lippmeier for providing the Repa benchmark results. We thank Gabriele Keller for improving a draft of this paper. The first author thanks Microsoft Research, Cambridge, for hosting him as an intern during the writing of this paper and to further improve the GHC's LLVM backend.

References

[1] clang: A C language family frontend for LLVM. `http://clang.llvm.org/`, 2010.

[2] LDC: LLVM D Compiler. `http://www.dsource.org/projects/ldc`, 2010.

[3] llvm-lua, jit/static compiler for lua using llvm on the backend. `http://code.google.com/p/llvm-lua/`, 2010.

[4] Openjdk - zero-assembler project. `http://openjdk.java.net/projects/zero/`, 2010.

[5] Unladen Swallow: A faster implementation of Python. `http://code.google.com/p/unladen-swallow/`, 2010.

[6] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[7] P. Codognet and D. Diaz. wamcc: compiling Prolog to C. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 317–331, 1995.

[8] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, Apr. 2007.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4.

[11] A. Dijkstra, J. Fokker, and S. D. 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.

[12] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93*, volume 28(6), pages 237–247. ACM Press, 1993.

[13] A. Gräf. The Pure programming language. `http://code.google.com/p/pure-lang/`, 2009.

[14] F. Henderson, Z. Somogyi, and T. Conway. Compiling logic programs to C using GNU C as a portable assembler. In *Proceedings of the ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages*, 1995.

[15] R. Hickey. The Clojure programming language. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. doi: http://doi.acm.org/10.1145/1408681.1408682.

[16] E. International. *Standard ECMA-355 - Common Language Infrastructure (CLI). Technical Report 4th Edition*. ECMA International, June 2006.

[17] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*, Sept. 2010.

[18] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, December 2002.

[19] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75. IEEE Computer Society, 2004. ISBN 0-7695-2102-9.

[20] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 1999. ISBN 0201432943.

[21] B. O'Sullivan. criterion: Robust, reliable performance measurement and analysis. `http://hackage.haskell.org/package/criterion`, 2010.

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

[23] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), 1992.

[24] S. L. Peyton Jones, N. Ramsey, and F. Reig. C--: A portable assembly language that supports garbage collection. In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 1–28. Springer-Verlag, 1999. ISBN 3-540-66540-4.

[25] N. Ramsey, J. Dias, and S. Peyton Jones. Hoopl: Dataflow optimization made simple. In *ACM SIGPLAN Haskell Symposium 2010*. ACM Press, 2010.

[26] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007.

[27] D. Tarditi, P. Lee, and A. Acharya. No assembly required: compiling Standard ML to C. *ACM Lett. Program. Lang. Syst.*, 1(2):161–177, 1992. ISSN 1057-4514.

[28] The LLVM Team. The LLVM compiler infastructure: LLVM users. `http://llvm.org/Users.html`.

[29] J. van Schie. Compiling Haskell to LLVM. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2008.