Evolution of the Spineless Tagless G-Machine

Armin Bernstetter

Seminar Funktionale Programmierung
Julius-Maximilians-Universität, Würzburg

Abstract. The spineless tagless G-machine (STGM) is an abstract machine that is located at the core of the Glasgow Haskell Compiler GHC. Since its creation at the start of Haskell development in early 1990s it has undergone several significant changes. This work aims at showing the evolution of the STGM and overall at providing insight in the workings of the most widely-used Haskell compiler GHC.

1 Introduction

This work provides an insight in the compilation process of the lazy, purely functional programming language Haskell. For this we take a look inside the Glasgow Haskell Compiler, today the most used compiler for Haskell [citation needed]. Located at its core is the *Spineless Tagless G-Machine*, an abstract machine used as a bridge between high level code and machine code.

Described in detail in the 1992 paper *Implementing Lazy functional languages on stock hardware: the Spineless Tagless G-machine* [4], the STGM has undergone several significant changes since then. Two papers highlighting these changes are *Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages*[7] and *Faster Laziness Using Dynamic Pointer Tagging*[8]. The former introducing the switch from the *push/enter* evaluation method to the *eval/apply* (see section BLA), the latter introducing dynamic pointer tagging which revokes the "tagless" part in the name of the STGM.

Section 2 provides basic information about Haskell and compilers in general.

Section 3 describes GHC, the *Glasgow Haskell Compiler* which is the most widely-used Haskell compiler. This section introduces the building blocks that GHC consists of.

Section 4 takes a more in-depth look into the *Spineless Tagless G-Machine*, an abstract machine that stands between Haskell code and assembly code in GHC's compilation process.

Section 5 concludes with a retrospective overview of STGM and its changes throughout the last 30 years.

2 Basics

This section introduces basics about compilers, Haskell and functional programming in general.

2.1 Haskell

Haskell is a purely functional programming language that emerged during the late 1980s and early 1990s. It was created with the goal of finding a common functional language to improve interactivity and exchange between programmers and researchers since, at the time, many lesser known functional programming languages existed. A committee consisting of Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler and others was created and met several times until in 1990 the Haskell 1.0 Report was published [3].

2.2 Compilers

A compiler is a software system consisting of several phases, that translates programs from a higher-level language to machine code. [9]

In general, these phases are *lexical analysis*, *syntactic analysis or parsing*, *semantic checking* and *code generation*. [9]

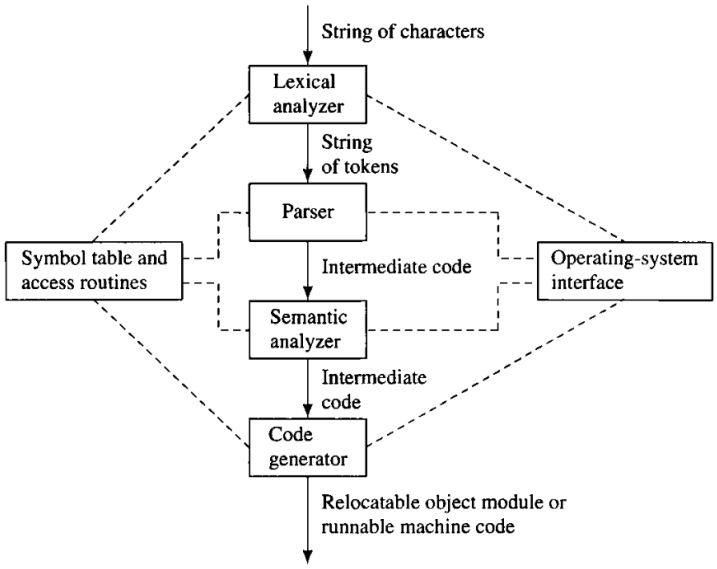

Fig. 1. General illustration of the phases of a compiler taken from Muchnik [9].

Lexical Analysis analyzes the character string and produces errors in case any part of the program string is not parseable into legal tokens. Legal in this case refers to tokens that are members of the vocabulary of the respective programming language.

Syntactic Analysis/Parsing parses the program into an intermediate representation. An example would be a parse tree accompanied by a symbol table containing information on identifiers used in the program and their attributes. This phase may also produce error messages if syntax errors are detected.

Semantic Checking examines the program for static-semantic validity. This phase takes as input the intermediate representation and determines whether the program satisfies the requirements for the static-semantic properties of the source language.

Code Generation finally transforms the intermediate representation into machine code which can then be executed.

These phases are often complemented by additional steps in many compilers, the Glasgow Haskell Compiler being one of those (see Section 3).

2.3 Abstract Machines

This section describes the concept of abstract machines. They bridge the gap between high level source code and machine code by providing an intermediate language stage for compilation. Abstract machines are located in the space between the two extremes of being a small intermediate language and being a model for a real machine that is yet to be built [1].

Abstract machines execute programs step-by-step in a loop. This execution loop iterates over a sequence of instructions often using a stack and register with the program counter as a special register pointing at the next instruction. [1]

Abstract machines introduce an additional layer of abstraction to the implementation of compilers for programming languages.

Examples for such abstract machines or related concepts are the Java Virtual Machine and the Spineless Tagless G-Machine which is the focus of this work.

3 GHC

link ghc 7.8.1

link ghc 8.6.5

The Glasgow Haskell Compiler, named after the city where its development began, is the most widely-used Haskell compiler [8]. It is the de facto default compiler for Haskell and is shipped with the *Haskell Platform* downloadable at `https://www.haskell.org/`.

Figure 3 illustrates the different phases program code is passed through during compilation. In contrast to the general compiler structure shown in figure 2.2, GHC has several additional intermediate steps.

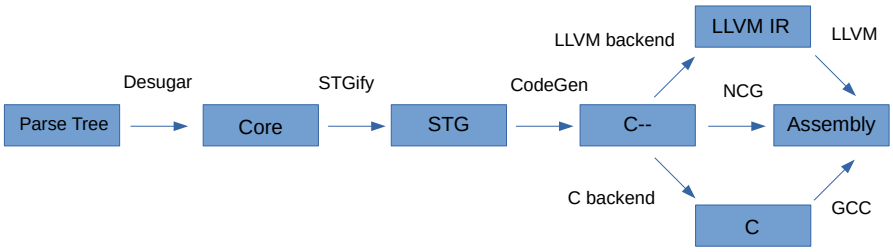

Fig. 2. GHC's compilation phases. Graphic based on a depiction from the official GHC gitlab repository [2]

3.1 Core Language

The core language is a variant of Haskell in where all syntactic sugar is removed and resolved e.g. the do-notation or type aliases. It consists of Haskell's central data types and is a small, explicitly-typed variant of the typed lambda calculus System F [2] which is called System FC [10].

3.2 The STGM Language

The STG language is a purely functional programming language on its own that is used as an intermedi

syntax details?

heap object layouts?

Move to "in depth" section?

3.3 *C − −*

[5] *C − −* is a programming language developed by Simon Peyton-Jones as a portable backend-language for compilers. Its name references C and C++. Where C++ can be seen as an extension of the C language, C– is to be thought of as a reduction to a smaller core language. C– in general is made for being generated by compilers and not for being written by programmers.

Why is it needed? C and Stack stuff?

3.4 Backends

GHC supports the generation of assembly machine code via several backends.

[2] `https://gitlab.haskell.org/ghc/ghc/wikis/commentary/compiler/generated-code`

C Backend The C backend uses the *GNU Compiler Collection* GCC but is deprecated

Native Code Generator The Native Code Generator is the default way used in GHC.

LLVM LLVM is a modern portable compiler framework that was developed as an alternative to the classic GCC toolchain. Its acronym stands for *Low Level Virtual Machine* [6]

Using LLVM in GHC results in similar compilation performance as the NCG but can lead to faster performing executables.

4 STGM in depth

Vielleicht zuerst einfach nur den Status von 1992 zeigen und dann die Änderungen? (Push/enter -¿ eval/apply und tagless -¿ dynamic pointer tagging) -¿ Sehr gute Idee!

[4]

Spineless Spineless refers to the way the code is represented on the machine.

STG programs are not represented as a tree but as a graph. Therefore, in memory a STG program is not a contiguous block of memory but smaller parts of the graph that reference each other.

Tagless SPJONES 1993 Seite 12 The term tagless refers to the way the STG-machine evaluates a heap closure.

[EVERYTHING FROM THE INTRODUCTION OF THE POINTER TAGGING PAPER]

4.1 Function application

in SPjones 1993 Seite 14/15

Function calls in a lazy functional languages with currying and partial application require special mechanisms in compilers. “Currying” is one of the core principles of lazy functional languages

[INSERT DETAILS ABOUT CURRYING]

HOW TO INCLUDE THE EVALUATION TRACE?

Push/Enter

Eval/Apply

4.2 Dynamic Pointer Tagging

In 2007, Marlow et al.[8] found that Haskell programs compiled by GHC show mispredicted branches on modern processors. This led to a re-examination of the "tagless" aspect of the STGM. The result were significant performance improvements.

5 Conclusion

This work

References

1. Diehl, S., Hartel, P., Sestoft, P.: Abstract machines for programming language implementation. Future Generation Computer Systems **16**(7), 739–751 (2000)
2. Girard, J.Y.: The system f of variable types, fifteen years later. Theoretical computer science **45**, 159–192 (1986)
3. Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of haskell: being lazy with class. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. pp. 12–1. ACM (2007)
4. Jones, S.L.P.: Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. Journal of functional programming **2**(2), 127–202 (1992)
5. Jones, S.P., Ramsey, N., Reig, F.: C-—: A portable assembly language that supports garbage collection. In: International Conference on Principles and Practice of Declarative Programming. pp. 1–28. Springer (1999)
6. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. p. 75. IEEE Computer Society (2004)
7. Marlow, S., Jones, S.P.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. In: ACM SIGPLAN Notices. vol. 39, pp. 4–15. ACM (2004)
8. Marlow, S., Yakushev, A.R., Peyton Jones, S.: Faster laziness using dynamic pointer tagging. In: Acm sigplan notices. vol. 42, pp. 277–288. ACM (2007)
9. Muchnick, S., et al.: Advanced compiler design implementation. Morgan kaufmann (1997)
10. Sulzmann, M., Chakravarty, M.M., Jones, S.P., Donnelly, K.: System f with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation. pp. 53–66. ACM (2007)