# 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* [9], 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*[12] and *Faster Laziness Using Dynamic Pointer Tagging*[13]. 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 some basics about

## 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 [7]. One of the foundations Haskell is based on is the principle of lazy evaluation. Lazy (or non-strict, or call-by-need) evaluation causes compiler development for lazy langauges to differ significantly from strict languages such as C.

## 2.2 Compilers

A compiler is a software system consisting of several phases, that translates programs from a higher-level language to machine code [14].

In general, these phases are *lexical analysis*, *syntactic analysis or parsing*, *semantic checking* and *code generation* [14]. Figure 1 shows an illustration of these phases and their transitions.

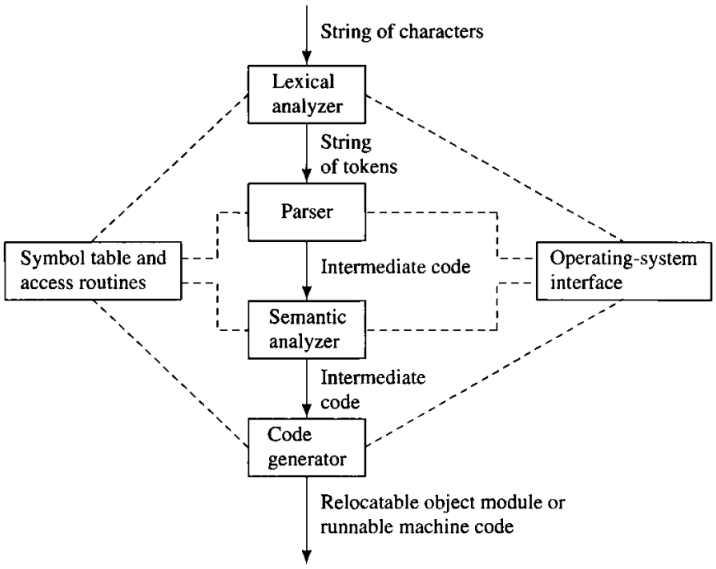


**Fig. 1.** General illustration of the phases of a compiler taken from Muchnik [14].

*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

Abstract machines bridge the gap between high level source code and machine code by providing an intermediate language stage for compilation. They are located in the conceptual 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 [3].

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. [3]

These 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

The Glasgow Haskell Compiler, named after the city where its development began in 1989, is the most widely-used Haskell compiler [13]. It is the de facto default compiler for Haskell and is shipped with the *Haskell Platform* downloadable at `https://www.haskell.org/`.

Figure 2 illustrates the different phases program code is passed through during compilation. In contrast to the general compiler structure shown in figure 1, GHC has several additional intermediate steps.

The Haskell source code is initially parsed and translated to a reduced *Core* language which is then again translated to the *STG representation*. A code generator (the *STGM*) generates `C--` code followed by three possible paths for finally generating machine code.

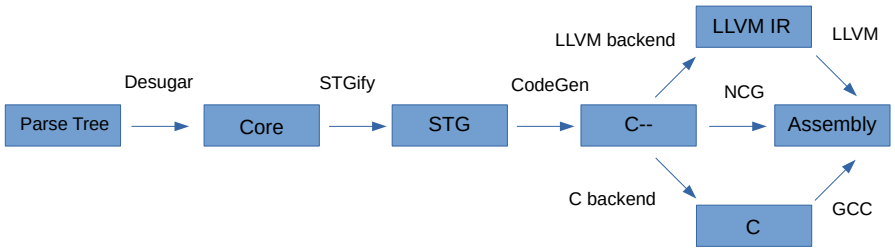


**Fig. 2.** GHC's compilation phases. Own graphic based on a depiction from the official GHC repository [2]

### 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 [5] which is called System FC [15]. Type checking is performed here, overloading is resolved and pattern-matching is translated into `case` expressions where each one performs only a single level of matching [9].

### 3.2 The STG Language

The core language is then translated to the *STG language*. This language is a purely functional programming language on its own that is used as an intermediate representation.

Some of its characteristics defined by Peyton Jones [9] are:

*All function and constructor arguments are simple variables or constants.*
*All constructors and built-in operations are saturated.*
*Pattern matching is performed only by* `case` *expressions.*

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

Each of the constructs in the language corresponds to an operational reading.

Each `let` expression allocates an object in the heap. The nature of heap objects is expanded on in section 4. A `case` expression forces evaluation of a sub-expression. A function application represents a tail call and a constructor application represents the return to a continuation.

| Construct | Operational reading |
|---|---|
| Function application | Tail call |
| `let` expression | Heap allocation |
| `case` expression | Evaluation |
| Constructor application | Return to continuation |

**Table 1.** Table taken from Jones 1993 [9]

Section 4 will go into further detail on the syntax of the intermediate STG language.

### 3.3 `C--`

A code generator (the STG machine) translates the STG language into `C--`. In fact, in Peyton Jones's 1992 paper [9] this step generated an intermediate representation called *Abstract C*. It was only several years later that `C--` came into being [10].

`C--` is a programming language developed by Simon Peyton-Jones as a portable backend-language for compilers. 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. [10]

### 3.4 Backends

GHC supports the generation of assembly machine code via multiple backends. In addition to a native code generator it offers a compilation route via LLVM and a route via C that has, however, been deprecated.

**Native Code Generator** The Native Code Generator is the default backend used in GHC and compiles `C--` directly to assembly code. The NCG is the fastest backend and produces well performing code. [1]

**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* [11].

Using LLVM in GHC results in similar compilation performance as the NCG but can lead to faster performing executables for some cases. In these cases - like

numeric, array heavy code - the penalty is a significant increase in compilation times, though. To use GHC with the LLVM backend, LLVM has to be installed on the respective system.[1]

**C Backend** The C backend uses the *GNU Compiler Collection* GCC but has bee deprecated since around GHC version 7.0 (2011) [1].

## 4 STGM in depth

The Spineless Tagless G-Machine is an abstract machine for non-strict functional languages. It was developed by Simon Peyton Jones as an alternative or successor to other such abstract machines like the *Three Instruction Machine (TIM)* by Fairbairn and Wray [4] and the *G-Machine* by Johnsson [8]. The STG machine transforms the incoming code from the core language to `C--` via the intermediate STG language.

**Closures** Heap objects can be either evaluated *head normal forms* (*values*) or unevaluated suspensions (*thunks*).

In the context of the STGM, the term *closure* refers to both values and thunks.

Such closures are represented as a pointer to a contiguous block of heap-allocated storage which consists of a *pointer* pointing to the static code, followed by the values of any free variables.

**Heap Objects** There are five kinds of heap objects in the STG language [12].

$FUN(x_1 \dots x_n \to e)$: A function closure $FUN$ has $n$ arguments and a body $e$. Such a function may be applied to more or fewer than $n$ arguments but still has an *arity* of n i.e. it is *curried*.
$PAP(fa_1 \dots a_n)$: A partial application $PAP$ represents a function $f$ that is applied to $n$ arguments $x_i$. The function $f$ is a $FUN$ object and its *arity* is strictly greater than $n$.
$CON(Ca_1 \dots a_n)$: A constructor object $CON$ represents a data value. The constructor $C$ is applied to $n$ arguments. This application is always saturated, that means the arity of $C$ is always equal to the number of arguments.
$THUNKe$: A Thunk is an unevaluated expression, also called *suspension*. A thunk is evaluated and overwritten with the value of $e$ when this value is needed.
$BLACKHOLE$: Is an object that is only used during the evaluation of a thunk. It replaces the thunk to prevent e.g. space leaks.

Listings 1 and 2 show an implementation of a `map` function in Haskell and its representation in the STG language respectively.

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Listing 1: A `map` function implemented in Haskell.

```
nil = CON Nil
map = FUN (f xs ->
        case xs of
           Nil -> nil
           Cons y ys -> let h = THUNK (f y)
                            t = THUNK (map f ys)
                            r = CON (Cons h t)
                        in r
     )
```

Listing 2: The map function from listing 1 represented in the STG language

**"Spineless"** The term "spineless" refers to the way the STG 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 and can be shared (common subexpressions).

The STGM is based on *graph reduction*, a concept first introduced in a Ph.D. dissertation by Christopher Wadsworth in 1971 [2]. Henderson et al. continued this concept in 1976 by presenting a different and lazy way to execute LISP programs [6] which was then again the basis for lazy functional programming languages including Haskell. Wadsworth suggested a graph-reduction strategy that used pointers to implement sharing of closures.

**"Tagless"** The term tagless refers to the way the STG-machine evaluates a heap closure. All heap objects have a uniform representation with a code pointer in their first field. This is true for both unevaluated suspensions and head normal forms. The STGM does not examine tag fields in those objects to decide how to treat them but rather makes a jump to the code pointed to. [9]

This concept is explained in an example by Marlow et al. [13]. Take for example the expression

```
f x y = case x of (a,b) -> a + y
```

The compiler of a lazy language has to ensure that `x` is evaluated before taking it apart into patterns. It pushes a continuation for the computation `a+y` onto the stack and jumps to the *entry code* for `x`. This jump to the entry code

is called *entering* the closure. If the closure is unevaluated, the entry code will evaluate it and return its value to the continuation, otherwise it will return immediately. A tag-ful approach would only enter the closure if it is not yet evaluated. This would be checked by performing an extra test on the tag i.e. the type of the closure. The benefit of a tagless approach is that any closure can be evaluated simply by entering it which is a simple and uniform process. [13]

### 4.1 Function application

Function calls in a lazy functional languages with currying and partial application require special mechanisms in compilers. Showing how function application is performed provides much insight into the compilation strategy of a language like Haskell.

*Currying* is one of the core principles of lazy functional languages such as Haskell. A function

```
f x y = x
```

has the type `a -> (b -> a)`. It can be seen as a function which takes one argument and returns a function which takes the second argument. An application like `(f 1 2)` is short for `((f 1) 2)`. Applying `f` to only one argument is a valid expression and is called *partial application*.

Compilers based on lazy graph reduction often compile function application using a strategy called the *push-enter model*. This model was initially used by the STG machine. The basic principle is that the argument(s) given to a function are *pushed* on an evaluation stack and then the function is tail-called (or *entered*)[9].

Consider the program shown in listing 3. This short program is taken from the code examples of the *Ministg* project. This project "is an interpreter for a high-level, small-step, operational semantics for the STG machine" by developer Bernie Pope. Its repository can be found on `https://github.com/bjpop/ministg`.

It offers functionality for tracing the evaluation steps which the STG machine takes to evaluate function application.

The program shown in listing 3 simply returns `true`. The function `const` takes two arguments, throws away the second and returns the first. The function `apply` takes a function `f` and another argument `x` and returns the function `f` applied to `x`. `twentytwo` is a constructor value that represents the integer *22* and `true` is a constructor that represents the boolean value `True`. The program `main` is a `THUNK` i.e. an expression, which is yet to be evaluated.

```
const = FUN(x y -> x);
apply = FUN(f x -> f x);
true = CON(True);
twentytwo = CON(I 22);
main = THUNK(apply const true twentytwo)
```

Listing 3: An example program `apply.stg` written in STG syntax.

Tables 2 to 8 show the evaluation trace for the program shown in listing 3 when using the *push-enter-model* of evaluation. The tables on the left show the arguments that are currently on the stack as well as the next expression to be evaluated. The tables on the right show the objects which are currently allocated on the heap and their variable names.

**Stack and Code**

| Stack | Expression |
|---|---|
| | `main` |

**Heap**

| Variable | Object |
|---|---|
| `apply` | `FUN(f x -> f_? x)` |
| `const` | `FUN(x y -> x)` |
| `main` | `THUNK(apply_2 const true twentytwo)` |
| `true` | `CON(True)` |
| `twentytwo` | `CON(I 22)` |

**Table 2.** The objects have been allocated on the heap and the expression `main` is next to be evaluated.

**Stack and Code**

| Stack | Expression |
|---|---|
| `upd * main` | `apply_2 const true twentytwo` |

**Heap**

| Variable | Object |
|---|---|
| `apply` | `FUN(f x -> f_? x)` |
| `const` | `FUN(x y -> x)` |
| `main` | `BLACKHOLE` |
| `true` | `CON(True)` |
| `twentytwo` | `CON(I 22)` |

**Table 3. THUNK:** `main` is a thunk and has therefore been replaced by a `BLACKHOLE` on the heap during its evaluation. `upd * main` which is currently on the stack will eventually update `main` with its value. The next expression to be evaluated is `apply_2` ..., a function of *arity* 2 which is given three arguments.

**Stack and Code**

| Stack | Expression |
|---|---|
| `arg const` | `apply` |
| `arg true` | |
| `arg twentytwo` | |
| `upd * main` | |

**Heap**

| Variable | Object |
|---|---|
| `apply` | `FUN(f x -> f_? x)` |
| `const` | `FUN(x y -> x)` |
| `main` | `BLACKHOLE` |
| `true` | `CON(True)` |
| `twentytwo` | `CON(I 22)` |

**Table 4. PUSH:** The arguments `const`, `true` and `twentytwo` have been pushed onto the stack. The next expression to be evaluated is `apply`.

**Stack and Code**

| Stack | Expression |
|---|---|
| `arg twentytwo` | `const_? true` |
| `upd * main` | |

**Heap**

| Variable | Object |
|---|---|
| `const` | `FUN(x y -> x)` |
| `main` | `BLACKHOLE` |
| `true` | `CON(True)` |
| `twentytwo` | `CON(I 22)` |

**Table 5. FENTER:** the function `apply` has been *entered* and returned the expression `const_? true`. These are the two arguments that `apply` "took" from the stack. `const_?` is of yet unknown arity.

**Stack and Code**

| Stack | Expression |
|---|---|
| `arg true` | `const` |
| `arg twentytwo` | |
| `upd * main` | |

**Heap**

| Variable | Object |
|---|---|
| `const` | `FUN(x y -> x)` |
| `main` | `BLACKHOLE` |
| `true` | `CON(True)` |
| `twentytwo` | `CON(I 22)` |

**Table 6. PUSH:** The argument `true` that `const_?` was applied to has been pushed onto the stack. The next expression to be evaluated is `const`.

**Stack and Code**

| Stack | Expression |
|---|---|
| `upd * main` | `true` |

**Heap**

| Variable | Object |
|---|---|
| `main` | `BLACKHOLE` |
| `true` | `CON(True)` |

**Table 7. FENTER:** `const` now has found the remaining two arguments it needed on the stack and returned the first of them, `true`.

**Stack and Code**

| Stack | Expression |
|---|---|
| | `true` |

**Heap**

| Variable | Object |
|---|---|
| `true` | `CON(True)` |

**Table 8. UPDATE:** the expression `true` has been evaluated and is the new value that the thunk `main` has been updated with. The computation is now finished.

### 4.2 Changes since 1992

The Glasgow Haskell Compiler and in consequence its several intermediate phases are in constant development. As a result, the Spineless Tagless G-Machine has undergone multiple changes since Simon Peyton Jones's paper in 1992 [9].

`C--` As mentioned already in section 3.3 before `C--` was developed its place in GHC's pipeline was taken by an intermediate representation called *Abstract C*

[TODO: more detail from [10]]

**Eval/Apply** In 2004, Marlow and Peyton Jones reevaluated the choice for the *push-enter* model for function application [12]. They found, that the *eval-apply* model, which had initially been discarded [9] actually delivered better compilation performance.

[TODO: Explain eval-apply (aber ohne trace)]

**Dynamic Pointer Tagging** In 2007, Marlow et al.[13] 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. Glasgow Haskell Compiler User's Guide (2019), `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/codegens.html`
2. Christopher, P.: Wadsworth. 1971. Semantics and Pragmatics of the Lambda Calculus. Ph. D. Dissertation. Oxford University
3. Diehl, S., Hartel, P., Sestoft, P.: Abstract machines for programming language implementation. Future Generation Computer Systems **16**(7), 739–751 (2000)
4. Fairbairn, J., Wray, S.: Tim: A simple, lazy abstract machine to execute supercombinators. In: Conference on Functional Programming Languages and Computer Architecture. pp. 34–45. Springer (1987)
5. Girard, J.Y.: The system f of variable types, fifteen years later. Theoretical computer science **45**, 159–192 (1986)
6. Henderson, P., Morris Jr, J.H.: A lazy evaluator. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages. pp. 95–103. ACM (1976)
7. 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)
8. Johnsson, T.: Efficient compilation of lazy evaluation, vol. 19. ACM (1984)
9. 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)
10. 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)
11. 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)
12. 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)
13. 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)
14. Muchnick, S., et al.: Advanced compiler design implementation. Morgan kaufmann (1997)
15. 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)