SE 464 - Lab 3
Architectural Analysis of the Glasgow Haskell Compiler

Collins Chung, 20390487
Chao Gao, 20374621
Alexander Maguire, 20396195
Shival Maharaj, 20378218

July 16, 2014

1 Introduction

For our project, we have decided to analyze the architectural implementation of GHC – the Glasgow Haskell Compiler. GHC is a widely adopted (and indeed the de-facto standard) Haskell compiler. It enjoys fantastic optimization support and better error-messages than any other Haskell compiler in existence.

GHC is noteable for being under continuous development over the last 20 years, which we consider makes it the ideal target for an architectural analysis.

2 System Functionality

GHC's primary goal is to serve as a high performance compiler for Haskell programs. The main user stories supported by GHC are as follows:

1. As a Haskell programmer, I want to compile Haskell programs into high performance code which runs on multiple platforms.
2. As a GHC developer, I want others to be able to share and re-use the GHC code as a basis for their own projects.
3. As a developer, I want to share and re-use packages with members of the Haskell community.
4. As a tools developer, I want to be able to use GHC to build tools, such as IDEs, that work with Haskell source code.

3 Quality Attributes and Scenarios

1. **Extensibility**: As a library developer, I want to write a compiler extension to allow GHC to perform domain-specific optimization of my code.
2. **Modularity**: As a researcher, I would like to be able to add experimental features and optimizations to GHC.
3. **Portability**: As a Haskell developer, I would like to be able to generate code to target different processors and platforms.

4 Architectural Views

1. **Process View**: This view deals with the dynamic elements of the system during runtime. In particular, it explains how system processes and threads are connected and how they communicate with one another. The process view is useful for addressing issues related to concurrency, scheduling, and performance.
2. **Phase View**: The phase view is a conceptual understanding of the individual stages of the compiler itself. It allows us to reason about how the phases (eg. parsing, typechecking, code generation, etc.) interact with one another. As such, it is closely related to the development view (described below).
   ¡¡http://www.aosabook.org/images/ghc/hscpipe2.png!!

3. **Development View**: The development view shows the system from a programmer's point of view by illustrating the organization of software modules, libraries, and units of development. Given that GHC is open source, this view is useful for programmers who wish to contribute to the compiler.

4. **Deployment View**: The deployment view displays how code is translated into hardware elements. It allows one to see on which physical elements a particular software element will be allocated to. This gives an engineer a useful view to gauge performance, security, or availability. Specifically for availability, it can help make the GHC more portable on different hardware environments.

5 Architectural Analysis

Extensibility: GHC allows library authors to define rewrite rules by rewriting the program during the optimization, so the programmers can extend GHC with domain-specific optimizations. The template method design pattern can be applied here to achieve such attribute. The template method pattern allows to redefine certain steps of an algorithm without changing the algorithm structure. In our case, GHC allows to apply transformations to certain expressions which depends on the rewrite rules, so the template method pattern can make GHC adapt to the changes without changing the entire structure.

Also GHC supports plugins by allowing programmers to write a pass that is inserted directly into compiler's pipeline. As well, GHC provides package system which allows users to manage libraries of Haskell code written by other people, and use them in their own programs and libraries. The plugin architecture can be applied to address the attribute because the architecture can easily add specific features to the existing core application to enhance the application's functionality. The plugin architecture allows GHC to support plugins and provide package system due to its high extensibility.

Both the template method design pattern and plugin architecture are related to GHC's development view.

Modularity: GHC allows researchers to make their own modifications to GHC to add new experimental features or optimisations. GHC contains a Haskell parser, abstract syntax, and type checker to allow people to build tools and infrastructure to understand Haskell source code. GHC is built as a library and a small Main module is linked to it to make the GHC executable itself. Also, GHC forces developers to make all the dependencies explicit which increases its modularity. Modular architecture is suitable in GHC's design to make it obtain such attribute. Modular architecture provides low coupling between the separate individual components which make the GHC component independent. Thus the researchers can experiment and optimize the GHC modules without affecting the application's entire functionality. The modular architecture is related to GHC's development view.

Portability: GHC has a block layer and the garbage collector is built on top of it to manage memory in units of blocks. The memory areas are repre-

sented as linked lists of blocks, so the GC does not have the overhead of fitting multiple resizable memory areas into a flat address space. This design has great portability because it needs very little support from the OS. The layered architecture can be used to address this attribute. In layered architecture, each layer represents a level of abstraction or specificity and it relies on lower-level services from the layer below it and provides high-level services to the layer above it. The garbage collector and block layer can be represented as two layers. The block layer is lower-level layer and garbage collector is higher-level layer. The layer architecture provides great portability since the lower-level layers can be replaced easily, in our case it's the operating system. The layer architecture is related to GHC's deployment view.

6 Key Weaknesses

The architecture of GHC is (perhaps unsurprisingly) very well designed, which is likely why the compiler has enjoyed such developmental longevity over the last two decades. As such, the authors of this paper have had difficulty determining major flaws in the architecture, and instead have decided to focus on some non-ideal technical necessities which have precipitated out due to a design-decision of the compiler itself.

GHC aims towards homogeneity of library and user-code, preferring to spend development time on global optimizations rather than tight optimizations of library-code. Such a strategy ensures that all code existing in user-land gains the full optimization strategies of the compiler, though this gain comes with a significant cost: non-deterministic inlining of function calls between modules.

In order to optimize function calls (functions are usually very small in Haskell) between modules, GHC considers it necessary to inline most of these calls – even across module boundaries. While this has significant performance gains, it implies an unstable Application Binary Interface (ABI) between subsequent compilations of the same code.

An unstable ABI, in practice, means that the entire source tree must be recompiled every time a high-level module needs to be compiled – leading to unseemly compile times. As a further consideration, the entire library source tree must be included with all GHC versions, since it is almost impossible to use highly-performant runtime linking of library code.

References

[1] Marlow, S., & Peyton-Jones, S., *The Architecture of Open Source Applications: The Glasgow Haskell Compiler*. http://lulu.com, Massachusetts, Volume 2, 1st edition, 2012.