Final Project Report
CS-421: Programming Languages and Compilers

Comparing Libraries for Generic Programming in Haskell

Instructor:

Mattox Beckman

Prepared by:

Himangshu Das hdas4@illinois.edu

Project Github link here

Comparing Libraries for Generic Programming in Haskell

Overview

Motivation

The motivation for this project stems from my extensive professional experience as a Java developer, where I have frequently utilized generics to write flexible, reusable, and type-safe code. However, my knowledge of the implementation details at the compiler level was limited. The paper "Comparing Libraries for Generic Programming in Haskell" provides an in-depth analysis of various Haskell libraries for generic programming, offering a unique opportunity to deepen my understanding of these concepts.

Although the paper provides a thorough explanation of the benchmark theory, it lacks detailed instructions on how to run the tests and generate the results. In my project, I have attempted to fill this gap by outlining the necessary steps and demonstrating the process.

Paper Citation:

Original paper: "*Comparing Libraries for Generic Programming in Haskell*" Published in Chalmers Research Portal

Rodriguez, Alexey & Jeuring, Johan & Jansson, Patrik & Gerdes, Alex & Kiselyov, Oleg & Oliveira, Bruno. (2008). Comparing Libraries for Generic Programming in Haskell. Haskell'08 - Proceedings of the ACM SIGPLAN 2008 Haskell Symposium. 44. 111-122. 10.1145/1411286.1411301.

Goals

The primary goals of my project were to -

1. Review and summarize the paper "Comparing Libraries for Generic Programming in Haskell."
2. Analyze the strengths and weaknesses of different Haskell libraries for generic programming based on the criteria established in the paper.
3. Document the findings that provide practical insights for Haskell developers and use it for practical purposes.

Broad Accomplishments

1. A thorough review and summary of the paper is done along with all the supplementary materials.
2. I couldn't find a GitHub repository for the original paper's code. To address this, I have created a public GitHub repository for my project, which can be used by both myself and others.
3. The paper was published in 2008, several of the libraries it references are now unsupported. In my project, I have made an effort to update and clean up these libraries.

4. Compare the results from the paper with my local test. I haven't been able to run all the tests, but I have listed the ones I tested below.
5. Detailed documentation of findings and practical examples demonstrating the use of these libraries.

Implementation

The paper highlights the proliferation of generic programming libraries in Haskell and the need for a systematic comparison of different libraries, and while doing so it develops a set of benchmarks for such comparison.

Criteria for evaluation:

The paper considers following criteria for creating the benchmark -

1. **Types**: There are two things that are considered-
    a. **Universe Size**: The more types a generic function can be used on, the bigger the universe size for that library
    b. **Subuniverses**: Is it possible to restrict the use of a generic function to a particular set of data types, or to a subset of all datatypes? Will the compiler flag used on datatypes outside that subuniverse as a type error?
2. **Expressiveness**:
    a. **First-class generic functions**: Can a generic function accept another generic functions as an argument
    b. **Abstraction over type constructors**: In generic programming, defining an equality function (`==`) for different types is usually straightforward. However, creating a generalized `map` function that works for any container type (not just lists) is more challenging and isn't supported by all generic programming approaches. The ability to define such generalized functions is tested using `*gmap*` and `*crushRight*` in the code.
    c. **Separate compilation**: Can a datatype defined in one module be used with a generic function and type representation defined in other modules without the need to modify or recompile them?
    d. **Ad-hoc definitions for datatypes**: A generic function can include specific behavior for particular data types while handling others generically, tested by the `selectSalary` function.
    e. **Ad-hoc definitions for constructors**: Specific constructors can have custom definitions while others are handled generically, tested by the `rmWeights` function.
    f. **Extensibility**: A generic function can be extended with non-generic cases in different modules, tested by extending `gshow` for lists.
    g. **Multiple arguments**: Functions can take more than one generic argument, like a generic equality function.
    h. **Multiple type representation arguments**: Functions can be generic over multiple types, such as a generic `transpose` function (evaluation in progress).

    i. **Constructor names**: The approach can provide the names of constructors applied in a generic function, tested by `gshow`.
    j. **Consumers**, transformers, and producers: The approach can define generic functions that consume, transform, or produce data, exemplified by `*gshow*`, `*updateSalary*`, `*gmap*`, and `*gfulltree*`.
3. **Usability**: How convenient a library is to use, efficiency, quality of library distribution, portability.
    a. **Performance**: Compares running times for test functions for different libraries.
    b. **Portability**: How easier it is to port functions between different Haskell compilers
    c. **Overhead of library use**: This measures the additional programming effort required when using the generic programming library, including automatic generation of structure representations and instantiating generic functions.
    d. **Practical aspects**: Considers whether there is an implementation, its maintenance status, and the quality of its documentation.
    e. **Ease of learning and use**: Some libraries have complex implementation mechanisms, making them more difficult to learn and use.

Evaluation Summary:

This table below summarizes the evaluation of nine Haskell libraries as described in the paper. It shows that libraries like Spine, EMGM, and Smash generally support more features than others, excelling in areas like universe size, first-class generic functions, and performance. However, they might also introduce more overhead in terms of structure representations and work required for function instantiation/definition. Libraries like SYB and SYB3, while more limited in features, are easier to learn and use.

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate
Universe Size									
Regular datatypes	●	●	●	●	●	●	●	●	●
Higher-kinded datatypes	●	○	●	●	●	●	○	●	●
Nested datatypes	●	○	●	○	●	●	●	◐	●
Nested & higher-kinded	●	○	○	○	●	◐	○	◐	○
Mutually recursive	●	○	●	●	●	●	●	◐	●
Subuniverses	○	●	○	○	○	●	●	●	○
First-class generic functions	●	○	●	●	●	◐	●	◐	○
Abstraction over type constructors	●	◐	○	○	◐	●	●	●	○
Separate compilation	●	●	●	●	○	●	●	●	●
Ad-hoc definitions for datatypes	○	◐	●	●	○	●	●	●	●
Ad-hoc definitions for constructors	◐	◐	●	●	●	●	●	●	●
Extensibility	○	◐	○	●	○	●	●	●	○
Multiple arguments	●	●	◐	◐	●	●	●	◐	○
Constructor names	●	●	●	●	●	●	●	●	○
Consumers	●	●	●	●	●	●	●	●	●
Transformers	●	●	●	●	●	●	●	◐	●
Producers	●	●	◐	◐	◐	●	●	◐	○
Performance	◐	◐	○	○	◐	●	◐	●	●
Portability	●	○	○	○	○	●	○	○	●
Overhead of library use									
Automatic generation of representations	○	○	●	◐	○	○	◐	○	●
Number of structure representations	4	1	2	2	3	4	4	8	1
Work to instantiate a generic function	◐	●	●	●	◐	◐	◐	●	●
Work to define a generic function	◐	●	●	◐	●	●	◐	●	●
Practical aspects	○	◐	●	◐	○	○	●	○	●
Ease of learning and use	◐	●	○	○	●	◐	○	○	●

● Supported criterion
○ Unsupported criterion
◐ Partially supported criterion or unusual programming effort required

Components of the Code

The original code is copied from the paper link here. For my project I have imported the code to my Github link here.

The readme file at the root of the github repo contains the instructions to run the test and generate the report.

1. Summary Report: Contains a detailed summary of the paper and key findings.
2. Comparative Analysis: Documents the strengths and weaknesses of each library.
3. Practical Examples: Includes code snippets demonstrating the usage of various libraries.

Tests

All the tests are done using the file comparison/test.hs file. This was already a part of the original paper, in my project I have tried to run this on my local system. There have been some modifications also in terms of systems setup and dependencies, which are already committed to my Github repo.

Following are the instructions to run the tests -

1. Ensure Prerequisites:
    a. Ensure you have GHC (Glasgow Haskell Compiler) installed.
    b. Make sure all necessary Haskell libraries are installed (`base`, `directory`, `process`, `regex-compat`, etc.).
2. Set Up the Environment:
    a. Create an output directory named `out` where the results will be stored.
3. Compile and Run Tests:
    a. Navigate to the directory containing `test.hs` and execute the script using `runghc`. This command will compile and run the tests against the specified libraries and programs.
    b. Use the `--all` option to run tests on all libraries and programs. `runghc test.hs –all`
4. Generate Expected Output Files:
    a. If you need to generate expected output files for comparison, use the `--expected` flag followed by the library name and program name. This step is crucial for creating a baseline for future comparisons. runghc test.hs --expected LIGD TestSelectSalary
5. Analyze the Output:
    a. The results of each test, including compilation statistics and runtime outputs, will be stored in the `out` directory.
    b. Review the generated `.out` files for test results and `.compilestats` files for compilation statistics.

Listing

My code is available in this Github repo. Original paper code is listed here. Although the code is taken from the original paper, I made several modifications to ensure the tests ran successfully. This included updating dependencies to match the current Haskell ecosystem and configuring the environment to accommodate the necessary libraries and compiler options. These adjustments were essential for resolving module import issues and ensuring compatibility with modern GHC versions.

Summary of Findings

The findings in the original paper suggest that no single library excels across all criteria. Libraries that perform well in one aspect often struggle in others. For instance, extensible libraries like SYB3 require more boilerplate code than non-extensible ones and support a smaller universe size. On the other hand, EMGM, while providing extensible functions, complicates the definition of higher-order generic functions.

The choice of the best generic programming library depends on specific use cases, some of the specifics described below -

1. **Transformation Traversals**: SYB and Uniplate are recommended for their support for mutually recursive datatypes and ease of defining generic functions. Uniplate is ideal for monomorphic functions, while SYB3 and RepLib offer extensible traversals with some additional effort.
2. **Container Operations**: EMGM and RepLib are suitable for abstraction over type constructors and handling ad-hoc cases. Smash is an option, albeit with higher user effort due to numerous representations.
3. **Serialization**: EMGM and RepLib are again recommended for their support of constructor names, producers, ad-hoc cases, and mutual recursion. SYB, SYB3, and Smash are viable alternatives for those willing to adopt different APIs for producer functions.

To conclude, the selection of a generic programming library should be based on the specific requirements and scenarios of the application at hand.