Interim Report

for

Exploring Programming Language Design via Standard Language Semantics and Haskell Implementations

by

Aaron Win

Supervised by

Prof. Steve Reeves

COMPX591-21X(HAM)
University of Waikato
June 4, 2021

Contents

1 Introduction

Standard ways of giving meaning to programming languages (and then based on this writing compilers/interpreters for them) are well established. This project is open-ended in that it comprises taking a simple imperative language and its semantics and implementation as a starting point, and adding sophistication (methods, types etc,) incrementally, and using these various definitions to explore design alternatives for the language (call by name, call by reference, call by value, static binding, dynamic binding etc,).

One objective is to make these different designs as elegant and simple as possible. There might also be scope to include consideration of "optimisations" of the semantics of the implementations, along with (informal) proofs that optimisation does not change the semantics itself, only its "efficiency" when programs are run

2 Background

The study field of programming languages has been explored and expanded on by a range of programming experts for decades. Many such experts have developed their own philosophies for the problem of how to define a language formally and efficiently. While these philosophies have a tendency to use terminologies in common such as state, memory, value, etc, there is some interesting variety in how they perceived these terminologies and put them into practice by rebuilding certain language implementations around their own definitions. For example, a chosen standard on what truly constitutes a specific domain will dictate the way in which pertinent semantic clauses and functions are written with respect to items belonging to that domain. The level of strictness applied to definitions also differs between each standard, as it has been observed that in some cases the writing of looser definitions may either increase a program's flexibility or make it vulnerable to new bugs.

Over the years, some approaches to designing a language have proven objectively superior to others, as following them have given rise to the production of comparatively more efficient forms of tools for defining languages e.g., syntax parsers and interpreters. There are also several different ways in which the efficiency of such a tool can be evaluated, including but not limited to: time taken to compile, data usage, and ease of tasks such as debugging or type checking. It is perfectly possible to take a less efficient implementation and improve on it using a distinguished standard without affecting its semantics.

Perhaps the best way to observe (and understand) the effects that formalisation and optimisation of a programming language has on its efficiency is by performing the task oneself with a particular standard in mind. It is to be expected that with this hands-on experience comes a newfound insight into the optimisation process and the ways in which it improves the language implementation. Anyone wishing to study the field of programming languages would certainly benefit from personally optimising one while following a set of principles such as in the book: *The denotational description of programming languages – an introduction* by Michael J.C. Gordon. [1]

One way of compiling and interpreting programs is via functional programming, such as with *Haskell*; [2] a functional programming language known for performing 'lazy evaluation' in all its computations. *Haskell* can be used to develop tools for parsing/compiling and interpreting other languages, and just like tools built by other means, these tools may be modified to be more cost effective by some measure.

3 Approach

The starting point for the project was a set of *Haskell* [2] programs for a very basic programming language known as TINY. Between them the programs are able to compile a string of TINY code by parsing it into *identifiers*, *expressions* and *commands*, and interpret them correctly using direct semantic functions. Prior to modification, the programs were thoroughly tested to ensure no coding (structural, syntax, or other) errors would be carried to later versions.

From there, the approach to solving the problem outlined in the introduction has been to read through Gordon's book *The denotational description of programming languages – an introduction* [1], which was mentioned in the previous section, and apply the standard he described throughout to the original programs piece by piece. Most if not all milestones in the project have been about learning about different concepts from this book and applying them to the programs, reconstructing them such that their designs fit its standard. Progress has been made by working through this book, slowly gaining sufficient comprehension of the underlying theory and techniques and using these to formalise the semantics and implementations of these programs.

The main goal has been to get as far through the book as possible in the allotted time, so that the *Haskell* code becomes increasingly closer to fitting the book's model until the final deadline. There will be many months for the project, which will allow it to be done in a steady fashion.

4 Progress

By this point in the project, steady progress has been made and all deadlines for past tasks have been met. Listed below are the results and achievements thus far:

- Learned about the denotational description of standard semantics presented in Chapter 4 of Gordon's book. [1]
- Written standard and auxiliary functions in TINY, including: *cond*, *result*, *donothing*, *checkNum*, and *checkBool*.
- Was unable to write *'star'* function used for sequencing other functions (its purpose is that an error found in either argument function becomes the output of the *'star'* function which allows it to terminate immediately). This was due to software limitations, as a solution compatible with *Haskell*'s type system (which does not allow a function to have several different types) and the current program (which has different types of error for *expressions* and *commands*) did not seem to exist. However, this was not too much of a concern as the clauses using the *'star'* function would have to be redefined later on and their new iterations would not use it anymore.
- Successfully implemented standard semantics using standard and auxiliary functions. This involved rewriting all clauses as per Gordon's description (although without the *'star'* function to call they had to deal with errors themselves which required them to be written more complex).
- Gained understanding of the concept of continuations and the benefit they provide for a language interpreter.
- Successfully implemented continuation semantics i.e., the program has been modified so that all semantics clauses are written in terms of continuations. The advantage of this is that errors are not carried to the rest of the program. While it makes no noticeable difference during run time, it makes the program more efficient as certain clauses no longer have to work with errors.
- Removed the output from the state (instead the output is directly made a part of the end result), and set the end result to be a string of values followed by either *stop* (end of program) or *error* (something has gone wrong in the program so ignore the continuation and halt). This also required modification of the *[output E]* command.
- Written definitions for *location*, *store* and *environment*.
- Split *value* domain into three new domains: *storable value*, *denotable value* and *expressible value*.

- Studied and introduced the *declaration* semantic domain.
- Written rough drafts of TINY semantic functions in terms of new domains. These were overwritten before being refined.
- Written rough drafts of continuation transformation functions, including: *cont*, *update*, *ref*, and *deref*.
- Studied and introduced L and R values.
- Learned about the usage of *procedures* and *functions*.
- Implemented SMALL syntax parsing, introducing the syntactic domains of: *basic constant*, *operator*, *program*, and *declaration*.
- Introduced new semantic domains such as *basic value* and *file*.
- Written rough drafts of all SMALL semantic clauses, including new L and R value functions.

5 Plan

Very little has changed in terms of the plan, as the progress made so far is similar enough to what was anticipated and no unexpected setbacks have occurred. Shown below is the project plan for Trimester B:

Project Task	Due Date
Complete interpreter for SMALL	6 July
Implement escape and jump semantics	20 July
Implement procedure semantics (without parameters)	27 July
Implement procedure semantics (with parameters)	6 August
Conference abstract	13 August
In-class presentation	20 August
Conference presentation	25 August
Implement dynamic binding	31 August
Implement call by reference	10 September
Implement call by value	21 September
Implement array and record semantics	28 September
Implement file semantics	12 October
Implement new iteration semantics	22 October
Final dissertation	26 October

These tasks (excluding presentations and submissions) should be completed in the same way as previous tasks: by continuing to read through Gordon's book [1] to understand new concepts and employ them in modifications in the design of the programs.

References

[1] Gordon, M. J. (1987). *The denotational description of programming languages: An introduction.* New York: Springer.

[2] Haskell Language. (n.d.). Retrieved from https://www.haskell.org

A Brief

Shown below is the original brief of the project, borrowed from the previously submitted project proposal:

The main purpose of this project is to develop new understanding of how a programming language may be designed, in particular with regards to its semantics, and how a design might be improved on. The project itself will involve the formalisation and optimisation of simple imperative languages in *Haskell* (starting with a basic parser and interpreter) by first improving the efficiency of their semantics and implementations then enhancing the sophistication of their designs by adding more complex features such as methods. The intended outcome is a set of more formal versions of these languages with the same semantics but higher efficiency in their performance.

All construction/modification of code will be carried out using principles and theorems borrowed from Michael J.C. Gordon's book: *The denotational description of programming languages – an introduction.*