Project Proposal

for

Exploring Programming Language Design via Standard Language Semantics and Haskell Implementations

By Aaron Win

Supervised by Prof. Steve Reeves

March 19, 2021

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*.

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 will comprise 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 will be 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.

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 denotational description of programming languages – an introduction*.

One way of compiling and interpreting programs is via functional programming, such as with Haskell; 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.

Planned Approach:

The project will begin with a set of Haskell programs for a very basic programming language known as TINY. Between them the programs should be able to compile a string of TINY code by parsing it into identifiers, expressions and commands, and interpret them correctly using expression/command semantics. Ideally the interpretation would make use of the concepts which are crucial to processing syntax in TINY and other languages (state, memory, input, output, value, etc,). Prior to modification, the programs should also be thoroughly tested to ensure no errors are carried to later versions.

From here, most milestones in the project will be about applying different ideas from Gordon's book, *The denotational description of programming languages – an introduction*, to the programs, reconstructing them such that their designs fit its standard. Progress will be made by working through this book fragment by fragment, 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 will be 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.

For this project, the following will be needed:

- A working Haskell program for parsing a line of code in the language TINY into identifiers, expressions and commands;
- A working Haskell program for interpreting expressions and commands in TINY using semantic clauses and functions;
- Access to a Linux computer with *GHCi* installed, for testing all code;
- A copy of the book *The Denotational Description of Programming Languages*, by Michael J.C. Gordon, whose content includes many programming language concepts which it would be extremely advantageous to refer to throughout the project.

[Note: due to the nature of the project, this proposal does not include an evaluation section. Because it solely concerns learning about the standard way to define a language and implement it, there is nothing to judge against, meaning evaluation of the system to be built is not an issue.]

Conclusions:

The defining of programming languages is a widely explored field of study where enhancements can be made to implementations such that they are more formally and efficiently designed. Through rewriting an implementation thus redefining the semantics involved, one can learn about the philosophy and technique behind doing so as well as the overall advantages of the new version over the old one. As a result, a great deal of new awareness should be acquired by the participant, which is the basis for the project.

The project begins with a several programs for defining the language TINY. These will be slowly 'optimised' using Michael J.C. Gordon's standard, explained in detail in his book: *The denotational description of programming languages – an introduction*. This will involve revising existing code, adding new features, and extending the compiler to other languages. By the end of the project (28 October), it is anticipated that the new versions of the programs should satisfy most if not all language principles discussed in this book. Throughout the development, progress on it will be recorded for future presentations, reports, and other.

Schedule:

Test original programs	March 16
Write definitions for auxiliary functions	March 23
Implement *continuation* semantics for *command* and *expression*	April 9
Write new definition for *output*	April 16
Write definitions for *store* and *environment* domains	April 23
Split *value* domain into *storable*, *denotable* and *expressible*	11 May
In-class presentation	17 May - 20 August
Write definition for *declaration* domain	18 May
Implement *declaration continuation* semantics	25 May
Interim progress report	4 June
Implement basic *procedure* and *function* semantics	8 June
Create parser for SMALL	29 June
Create 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
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	28 October

References:

Because this project is more about exploration with a single standard rather than making comparisons between existing examples of implementations, only a small number of references will be needed at the start of the project.

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

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