A reflective language for the analysis of graph execution

Isaac H. Lopez Diaz

November 7, 2025

1 Introduction

This paper presents the design of a reflective programming language (PL) that reasons about its dataflow semantics. A PL is said to be reflective when is able to reason about itself.[7] It can be thought of as the process of converting data into a program. The inverse of this process, reification, can be thought of turning a program into data.[4] These two processes allow a programmer to see the contents of the current execution, much like debugging. However, unlike debugging, one can change the semantics of the language on-the-fly.[3]

The goal of the language is to better understand dataflow semantics. One such application would be on machine learning (ML) programs, since they rely on dataflow graph execution models[1] (the terms dataflow and graph execution will be used interchangeably). This dataflow graphs have their own semantics. This creates a burden on programmers since they'd be analyzing two programs with their own semantics, the host language like Python and the ML library like TensorFlow.[5]

Although, creating a new language may seem unnecessary, given that ML is having success, there is room for improvement. ML libraries have been extended permiting programmers to explicitly use dataflow semantics changing from the usual imperative semantics. However, this conversion from imperative to dataflow has proven to be challenging for programmers, primarily looking to optimize their code, leading to bugs or perfomance issues (the opposite of what the programmer intended to do).[9] In fact, these extensions are so pervasive that they even alter the execution and modification of state![1] Two surveys ([8], [9]) give a taxonomy of bugs which in some way correlates to programming languages. The cause of bugs can range from lack of features in a language (e.g. lack of type system guarantees) to a misunderstanding of the change of semantics from imperative to graph execution.

The paper is divided in three sections. First, reflective languages are explained and some small description of the one to be designed is given. Then, dataflow semantics are explained in the domain of ML programs. Additionally, the selection of the state of the language is justified. Finally, the design of the language is presented.

2 Reflective languages

Reflective languages started off with the notion of an infinite tower of interpreters. This means that you'd have an interpreter interpreting an interpreter, and so on. In order for a language to be reflective, it must have two properties: (1) the ability to reify its own interpreter, and (2) the ability to reflect on the reified interpreter.[4] This gives the ability to extend the language syntax and semantics.

Evaluating a language usually means consuming expressions and altering the environment, that is, the bindings of variables.[6] However, one can also keep track of two two more things: the continuation and the store. The continuation describes the control context, while the store describes the global state of the computation. Previous implementations have omitted the store [10]; for this language the store is needed, as explained on the next section.

The language to be designed would be much like a Lisp with the properties mentioned. The reason for choosing Lisp is because it takes care of syntactic details, that sometimes are just personal preferences, and in Lisp there is no distinction between code and data.[2] This allows for the ease of development and design of a reflective language, and manipulating the store of the language in creative ways.

3 Dataflow Semantics, Bugs and ML

4 The Design of the Language

The basis of our evaluator is the metacircular interpreter by

References

[1] Martín Abadi, Michael Isard, and Derek G Murray. A computational model for tensorflow: an introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–7, 2017.

[2] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs.* The MIT Press, 1996.

[3] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation: For a better understanding of reflective languages. *Lisp and Symbolic Computation*, 9(2):203–241, 1996.

[4] Daniel P Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 348–355, 1984.

[5] Mike Innes, Stefan Karpinski, Viral Shah, David Barber, PLEPS Saito Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, et al. On machine learning and programming languages. Association for Computing Machinery (ACM), 2018.

[6] Shriram Krishnamurthi. *Programming languages: Application and interpretation.* Brown University, 2017.

[7] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, 1984.

[8] Sebastian Sztwiertnia, Maximilian Grübel, Amine Chouchane, Daniel Sokolowski, Krishna Narasimhan, and Mira Mezini. Impact of programming languages on machine learning bugs. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis*, pages 9–12, 2021.

[9] Tatiana Castro Vélez, Raffi Khatchadourian, Mehdi Bagherzadeh, and Anita Raja. Challenges in migrating imperative deep learning programs to graph execution: an empirical study. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 469–481, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Mitchell Wand and Daniel P Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 298–307, 1986.