# A reflective language for the analysis of dataflow semantics

Isaac López Díaz, Humberto Ortiz-Zuazaga
*Department of Computer Science*
*University of Puerto Rico Río Piedras campus*
San Juan, Puerto Rico
isaac.lopez@upr.edu humberto.ortiz@upr.edu

***Abstract*—This paper presents the design of a reflective programming language (RPL) that has procedural semantics and dataflow semantics. It presents a methodology to implement the language and a partial implementation of it.**

***Index Terms*—reflective, semantics, procedural, dataflow**

## I. Introduction

### A. Problem and Motivation

It has been found that dataflow semantics create a burden on programmers, since they would be analyzing two programs with their own semantics, usually a host language like Python and an ML library like TensorFlow [5]. The goal of the language is to help programmers better understand and use these semantics. Our insight is that an RPL can help programmers understand dataflow in the context when used interchangeably with procedural semantics. This could allow the programmer to see what sort of graph is being built and how the state is being modified.

### B. Background and Related Work

An RPL is said to be reflective when it is able to reason about itself [7]. Reflective languages started as the notion of an infinite tower of interpreters, meaning that you would 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]. Reflection can be thought of as the process of converting data into a program, while reification, the inverse, is turning a program into data [4]. These two processes allow a programmer to see the contents, that is, the state, the environment, and the continuation of the current execution, much like debugging. However, unlike debugging, one can change the semantics of the language on-the-fly [3].

Although creating a new language may seem unnecessary, given that ML is having success, there is room for improvement. In fact, systems like TensorFlow are rooted in programming language concepts.[1] Today, 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 performance issues (the opposite of what the programmer intended to do, and the intention of the API) [9]. In fact, these extensions are so pervasive that they even alter the execution and modification of state [1] [9].

The impirical study by Tatiana et al. [9] give a taxonomy of bugs which in some case directly correlates to the misunderstanding of dataflow semantics being used along imperative semantics. Additionally, the survey by Sztwiertnia et al. present a list of bugs ranging from a lack of features in a language (e.g. lack of type system) to a misunderstanding of the change of semantics from imperative to dataflow semantics [8].

The RPL is designed based on the answers given in the survey by Castro et al. The research questions (RQs) in the study are the following:

- RQ1: What bug patterns and corresponding challenges are involved in writing reliable yet performant imperative (deep learning) DL code?
- RQ2: Which best practices and anti-patterns can be extracted from (RQ1)?

The paper is divided in three sections. The methodology followed to design the language, partial results, and a conclusion aiming at future work.

## II. Methodology

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 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. The reason being that the store will explicitly represent the graph that is being constructed while evaluating a program.

To implement the evaluator (or interpreter) of the language, the Lisp dialect Scheme is being used. The reason is because it takes care of syntactic details that sometimes are just personal preferences, and in Scheme there is no distinction between code and data [2]. This allows for the ease of development and design of an RPL, and manipulating the store of the language in creative ways.

What we want to achieve is the ability to reify and reflect between procedural and dataflow semantics. That is, evaluating some procedural expression gives you some graph, and some graph can be turned into some procedural program. This would allow one to see if the intention of the program written matches the expected output. Additionally, the reverse should be possible, if some output is the program one is thinking of writing.

## III. Partial Results

Currently, there is a partial implementation of the language. We give the skeleton of an RPL which is simple variant of Scheme, containing only: numbers, booleans, and conditionals. The program is able to get an expression and pass it to the Scheme evaluator and run it, reifying from the language and reflecting to Scheme.

```
(define (self-eval? e)
  (or (number? e)
      (boolean? e)))

(define (if? e)
  (eq? 'if (car e)))

(define (let? e)
  (eq? 'let (car e)))

(define (lambda? e)
  (eq? 'lambda (car e)))

(define (app? e)
  (pair? e))

;; meta-eval
(define (meta-eval expr cont)
  (cond
  ((self-eval? expr)
   (meta-apply expr cont))
  ((if? expr)
   (meta-apply 'eval-if expr cont))
   ...
  (else (error 'expr "unknown construct"))))

;; base
(define (base-eval proc-or-op expr cont)
  (let
  ((f (eval proc-or-op environment)))
    (f expr cont)))

;; eval if
(define (eval-if expr cont)
  (let* ((cnd (cadr expr))
  (then (caddr expr))
  (els (cadddr expr)))
 (if (meta-eval cnd cont)
 (meta-eval then cont)
 (meta-eval els cont))))
```

```
;; meta-apply
(define (meta-apply atom-or-func . rst)
  (cond ((self-eval? atom-or-func)
  ((car rst) atom-or-func))
 (else
  (let* ((expr (car rst))
  (cont (cadr rst)))
    (cont
         (base-eval
           atom-or-func expr cont))))))

;; repl/main
(define (blue)
  (display "> ")
  (let ((r (read)))
    (let ((ans (meta-eval r
           (lambda (ans) ans))))
      (display ans) (newline)
      (blue))))
```

Following the convention of dataflow semantics from Abadi, Isard, and Murray [1], the final language should have those semantics defined and implemented, then a programmer can apply the processes shown here of reifying and reflection to see the transformations between languages.

## IV. Conclusion

The goal is to present the problem programmers are currently having developing ML programs due to the semantics of graph execution models, and bring up as an idea a new language that is able to reason about such semantics. Also, a skeleton of a reflective language is given, from which to part from and build the complete version.

## 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.