Proposal: Gradual Typing for Octave Language[*]

University of British Columbia CPSC 311 Course Project

Ada Li
University of British Columbia
adali@alumni.ubc.ca

Yuchong Pan
University of British Columbia
yuchong.pan@alumni.ubc.ca

Kathy Wang
University of British Columbia
kathy.wang@alumni.ubc.ca

Paul Wang
University of British Columbia
paul.wang@alumni.ubc.ca

Abstract

Keywords gradual typing, Octave

1 Introduction

Static and dynamic type systems have their respective advantages. Static typing allows early error detection and enforces code style in a collaborative setting. It is, however, acknowledged that dynamic languages are better for fast prototyping. Over the past several decades, researchers in the programming language community have been working on integrating static typing and dynamic typing in a single language, having programmers control the level of type annotations. Gradual typing is a solution to combine the two type systems, proposed by [8]. It is of increasing interest in the programming language community and has been adopted by many programming languages, both in the industry and in the academia, such as Typed Racket [10], TypeScript [2] and Reticulated Python [11].

Octave is a scientific programming language with a dynamic type system. Because it has powerful matrix operations and is compatible with MATLAB scripts, it is widely used in statistics, mathematics and computer science communities for idea validation and fast prototyping. In this project, we propose a gradual type system for the Octave language to allow Octave programmers to annotate source code with optional type annotations, making Octave programs more robust and more suitable for production environments.

2 Overview and Plans

In this project, we will add a gradual type system to the Octave language. Because of the time limitation of the course, we will be focusing on the static semantics. Future work is discussed in Section 3. This means that our proposed gradual type system will mainly be used for static analysis and IDE tooling (e.g., code completion and refactoring, etc.), but not

for runtime type checking. Therefore, this project is similar to type hints in Python [1] and the type system of TypeScript.

2.1 How to Fulfill Background Research Report Milestone

We will be using the resources listed in Section **??** as starting points and guidelines for us to work from as we explore the articles that focuses on gradual typing. We will cover its value in everyday programming practice as well as cover the technical elements.

2.2 How to Fulfill Proof-of-Concept Milestone

For the proof-of-concept milestone, we will be focusing on implementing a static type checker for the gradually-typed lambda calculus (GTLC) by following [7]. The gradually-typed lambda calculus can be regarded a core of gradually typed programming languages, and implementing a static type checker for it will help us gain deeper understanding of gradual type systems. The following grammar defines the syntax of the gradually-typed lambda calculus [7].

variables x integers n blame labels l

basic types	B	::=	`int` | `bool`
types	T	::=	$B \mid (\to T\ T) \mid\ ?$
constants	k	::=	n | `# t` | `# t`
operators	op	::=	`inc` | `dec` | `zero?`
expressions	e	::=	$k \mid (op\ e\ l) \mid (\texttt{if}\ e\ e\ e\ l) \mid x \mid (e\ e\ l) \mid$
			$(\lambda(x)\ e) \mid (\lambda(x:T)\ e) \mid (e:T\ l)$

In addition, we are going to experiment various approaches to enforce type checking for matrix operations (e.g., matrix dimension checking), which are a significant component of the Octave language.

2.3 How to Fulfill Final Project Milestone

For the final project milestone, we would like to apply what we will have learned in the gradually-typed lambda calculus to the Octave language. To achieve this, we plan to implement a parser that converts Octave source code to abstract syntax trees in Racket, using parsing tools available in Racket.

---

[1]New in version 3.5.

For practical reasons, we are going to implement the most significant subset of the Octave language to demonstrate the features of Octave. After parsing, we will implement a static type checker with matrix operation support for abstract syntax trees.

2.4 How to Fulfill Poster Milestone

Our poster will emphasize the usefulness it has for users when dealing with matrices, which will entice our peers to learn more about the project as well possibly integrating it. This is particularly useful for students involved in mathematics and statistic courses as they will be using software like Octave frequently. We will be including diagrams on how this project works so students can easily understand without having to read a large amount of text. We will try to include an easy way for students to learn more about this project topic. Some ideas include compiling a list of topics on a single webpage and including that on our poster. Another idea would be creating a QR code for students to scan and receive more information.

2.5 How This Can Be a Low-Risk Approach

This project has a low risk because researchers in the gradual typing community have been exploring designs of gradual type systems for many years. There have been many publications in this field and other related fields, among which we have compiled a list of resources that may be of particular importance for our project, listed in Section **??**. For the language choice, Octave is an open source language compatible with MATLAB. This avoids copyright issues and ensures the audience of our project. In addition, there are many open source projects related to MATLAB/Octave parsing, which will help us with the parsing of Octave source code.

3 Future Work

Because of the time limitation of the course, this project will be mainly focusing on the static semantics of the proposed gradual type system for Octave. In the future, however, more work can be done to extend the proposed gradual type system. For instance, dynamic type checking could be added during the interpretation, properly tracking errors from source code. This can be done by translation into an internal cast calculus with the blame tracking [9]. In addition to that, type inference could be added to automatically deduce types at compile time. [5] introduces an approach to gradual type inference.

4 Related Work

There have been many research publications and projects on adding gradual type systems to an existing dynamic language. Diamondback Ruby (DRuby) is an extension to the Ruby language that combines static and dynamic typing in Ruby with constraint-based type inference [4]. Typed Racket

[10] recently began to support refinement and dependent function types as experimental features [3]. Other similar projects include Typed Lua [6] and Gradualtalk [1] which bring optional type systems to Lua and to Smalltalk, respectively. Our project would largely benifit from these projects.

Acknowledgments

References

[1] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2014. Gradual typing for Smalltalk. *Science of Computer Programming* 96 (2014), 52–69.

[2] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on Object-Oriented Programming*. Springer, 257–281.

[3] Racket Blog. 2017. Refinement Types in Typed Racket. http://blog.racket-lang.org/2017/11/adding-refinement-types.html

[4] Michael Furr. 2009. *Combining static and dynamic typing in Ruby*. Ph.D. Dissertation.

[5] Ronald Garcia and Matteo Cimini. 2015. Principal type schemes for gradual programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 303–315.

[6] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Typed Lua: An optional type system for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*. ACM, 1–10.

[7] Jeremy G Siek and Ronald Garcia. 2012. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*. ACM, 68–80.

[8] Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

[9] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[10] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 964–974.

[11] Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 45–56.