Report: 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

In this background report, we outline the general milestones and approaches that will be taken for our project of introducing gradual typing to Octave and provide brief motivation for the subject area. Further, we argue that the project is low-risk and interesting as a term project by showing relevant pieces of supportive literature and open-sourced projects and contend that there is much potential in future extensions.

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 [13]. 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 [15], TypeScript [4] and Reticulated Python [16]. We've decided to add gradual typing to Octave, as it is a language that suffers from inefficiencies of dynamic languages. Adding static types to Octave makes the scripts more robust and suitable for production environments, allowing programmers to evolve their programs gradually without refactoring to other languages.

2 Why Octave?

Octave is a scientific programming language with a dynamic type system. It is widely used in the statistics, mathematics and machine learning communities for idea validation and fast prototyping. It is an open source dynamically-typed scientific programming language compatible with MATLAB, so it has a lot of features in common.

Similarities Octave and MATLAB share are: matrices as fundamental data type, built-in support for complex numbers, built-in math functions with extensive function libraries, and extensibility of user-defined functions. This ensures that we have a large audience that will benefit from this. Also, since Octave and MATLAB share many similarities, we can use several supporting sources for MATLAB [3].

This paper [12] illustrates how Octave can benefit from introducing static type checking and shape analysis into a dynamically typed language. Specifically, static type and shape inferencing can improve the efficiency of the generated code. Functions and operators are highly overloaded in Octave, which can lead to bottlenecks; thus, the consequence/tradeoff for this flexibility is computational performance. The reason why is because the run-time system is responsible for type checking, array shape determination, function call dispatching, and handling possible run-time errors. In order to improve efficiency, these issues should be addressed at compile-time.

In addition, there are many open source projects related to MATLAB/Octave parsing, which will help us parse Octave source code. For instance, the matlab-parser is an ANTLRv4 parser for MATLAB programming language, which converts the MATLAB source code to a concrete syntax tree representation. Then the parse tree is processed to an abstract syntax tree and control flow graph [1]. Another MATLAB parser we can potentially reference, the mparser, is based on an ANTLR v3 grammar is also able to do source code translations to AST [2].

3 Why Gradual Typing?

In this section, we highlight several advantages of introducing a gradual type system to an existing dynamically-typed programming language, such as Octave in our proposed project.

3.1 Gradual Typing Incorporates Advantages from Static and Dynamic Typing

It is widely acknowledged that static and dynamic type systems have their respective benefits and drawbacks. For instance, a static type system allows mistakes in a computer program to be caught at an earlier stage, preventing fatal errors in production environments. In contrast, bugs in a dynamically-typed language can usually only be detected at runtime, requiring numerous unit tests to ensure the correctness of a piece of code. On the other hand, dynamically-typed languages are usually more flexible and accept various disciplines, making them more suitable for tasks like idea validation, fast prototyping and scripting. Static typing, however, enforces certain coding disciplines and abstraction, forcing programmers to consider design questions from very beginning. Therefore, both benefits and pitfalls co-exist in both type systems.

In a gradual type system, programmers are able to decide which regions of code in a computer program are statically or dynamically typed, making different pieces suitable for different stages and scenarios [13]. The Blame-Subtyping Theorem implies that statically-typed regions in a gradual type system cannot be blamed for type-casting errors and are therefore easy to analyze [14]. Hence, computer programs written in a gradually-typed languages take advantage of static typing in some regions and enjoy flexibility from dynamic typing in other regions, integrating benefits from both sides in a single program simultaneously.

3.2 Gradual Typing Can Be Implemented Efficiently

Statically-typed programs can be type-checked at compile time. However, partially-typed programs require runtime checks for runtime types of variables. The dynamic semantics of gradual type systems are usually defined by translation into an internal cast calculus like the Blame Calculus, which replaces implicit type casts with explicit type casts. For instance, the following steps illustrate the evaluation of a function application in GTLC based on the dynamic semantics defined in [13] (assuming succ has type int->int):

$$(\lambda(x:?)(succx))1$$

$$->(\lambda(x:?)(succ<?->int>x))<int->?>1$$

$$->succ<?->int><int->?>1$$

$$->succ1$$

$$->2$$

Because of explicit casts at runtime, people may be concerned by the performance of implementations of gradual type systems. Fortunately, Kuhlenschmidt et al. [11] present a space-efficient implementation approach that has competitive time efficiency on par with major fully statically- and dynamically-typed programming languages such as OCaml and Racket, and moreover that eliminates slowdowns for programs partially annotated with types.

3.3 Gradual Typing Enables Evolution of Programs

Due to flexibility of dynamically-typed programming languages, it is common in the industry that programs are initially developed in a dynamically-typed programming language for reasons like fast prototyping, idea validation and scripting. However, dynamically-typed programming languages usually expose programs to safety and security issues, which sometimes lead to fatal errors and even monetary losses in production environments. Furthermore, the runtime performance of dynamically-typed programs are usually not as satisfiable as that of statically-typed languages because static type systems are able to check many aspects of computer programs before they run. To resolve issues mentioned above, prototyping programs in the industry are usually refactored to a statically-typed languages, in order to reduce the possibility of errors as well as to enhance runtime performance.

This process, nevertheless, is time-consuming and error-prone. The gradual guarantee ensures that computer programs remains well-typed as long as type annotations are correctly added [14]. It is also guaranteed that the other direction, which is removing type annotations from a program does not break programs, is true in a gradual type system [14]. Therefore, with the aid of gradual typing, programmers are able to gradually evolve their programs from untyped to fully-typed with type annotations, without ever breaking the programs.

4 Related Projects

4.1 Static typing in Octave and MATLAB

Octave and MATLAB was designed to be easy to use and have syntax that follows closely with mathematical notation. However, with dynamic type checking, there are issues such as efficiency and the quality of the generated code. This paper [12] specifically looks at overloading in dynamically typed Octave programs. It introduces static typing for overloaded functions by making it explicit and restricting the input and output types of functions to match the actual function call. The disadvantages to this paper is that the resulting Octave syntax is not as close to mathematical notation as before. However, with the addition of this static typing, the system does not require any annotations from the user. Therefore, the user is not greatly impacted. Since this is now statically typed, it improves the computational performance of the system. This paper shows how mixing dynamic and static typing can improve the overall system for Octave.

Since we noted that MATLAB and Octave have many similarities, we can also examine related projects with MATLAB. As with Octave, MATLAB has the advantage of quick prototyping without the use of any static types. However, this is

a problem when it comes to reliability and efficiency. This paper [9] explains how they added typing aspects to MATLAB. They did this by adding the "atype" statement, which is a form of type annotation used to specify runtime types of variables, especially for function inputs and outputs. This is mainly used for input and output variables of a function, which helps with specifying, capturing, and checking dynamic types within the program. In addition, a weaver is used to convert their typed MATLAB to standard MATLAB so it still compiles normally. This is fairly similar to what we are trying to accomplish as we will be adding annotations to Octave for gradual typing.

Another way typing has been explored for MATLAB is telescoping. This paper [5] describes a way to create a telescoping system to generate high performance libraries from prototype MATLAB. In order to accomplish this, it needs to infer MATLAB types. This paper develops an approach of combining static and dynamic type inference. To start the inference, constraints are formed using a database of annotations containing one entry per procedure or operations. Unfortunately, this problem of inferring types is NP-hard. In order to make this an efficient algorithm, a few simplifying assumptions have to be made about the code. Once that is done, the problem can be reduced into an n-cliques problem which is NP-complete. One of the issues with this paper is that unanalyzed procedures will be ignored and those calls will not give any added information. This is a pitfall because all recursive calls are treated as unanalyzed procedures. We can see from this paper that typed MATLAB has its benefits but type inference can be an issue.

4.2 Gradual Typing Projects

Since the introduction of the theory of gradual typing, there has been many research projects and publications regarding adding an optional, gradual type systems to an existing dynamically-typed languages. In this section, we analyze several related projects and discuss how they are beneficial to our proposed project.

4.2.1 Diamondback Ruby (DRuby) [7]

Diamondback Ruby is an extension to Ruby with an optional static type system implemented by type inference. Programmers are able to annotate Ruby programs with type annotations, which will be checked by DRuby at runtime using contracts and blamed properly. It is also worth noting that DRuby is able to infer types on dynamic meta-language constructs, such as eval, through a combined static and dynamic analysis.

Although both DRuby and our proposed project both aim to add an optional type system to an existing dynamically-typed language, there are some differences between the two projects. For instance, because DRuby uses type inference to analyze Ruby programs, it requires static type checking for the entire program. That is, not all valid Ruby programs are valid DRuby programs. This design, however, violates the criteria for gradual typing, which says gradual type systems must accept both fully untyped programs and fully typed programs. Our proposed gradual type system to Octave, in contrast, allows Octave programs to be partially type-checked; that is, some regions of Octave programs are able to remain dynamic.

That being said, we can still learn much from the DRuby project. In the future, type inference could also be added to our proposed type system to Octave together with gradual typing. Garcia and Cimini [8] introduce a new approach to apply type inference on a gradually-typed language, admitting parametric polymorphism. In addition, the annotation syntax for DRuby is similar to informal documentation of Ruby, which facilitates programmers to annotate their Ruby programs. We will use this syntax as a reference when designing our annotation syntax for Octave.

4.2.2 Typed Racket [15]

Typed Racket is the gradual counterpart of Racket, enabling incremental addition of annotations for static type checking. It uses contracts to type-check at the boundaries of statically-typed regions and dynamically-typed regions at runtime. It also includes type inference, so many type annotations need not be done by programmers.

However, Typed Racket only supports gradual typing for whole modules. That is to say, a module can either be fully typed or be fully untyped, and boundaries of statically-typed regions and dynamically-typed regions can only exist at the boundaries of modules. This feature of Typed Racket guarantees that dynamic type checking can only happen at the boundaries of modules. It should be admitted that this design of Typed Racket improves runtime performance to some extent. However, it also introduces some inconvenience on the programmer side. As reported in [6], the limitations of module-level gradual typing requires to annotate all values with type Any in a module.

Typed Racket has recently begun to support refinement and dependent function types as experimental features [10]. Dependent types are first introduced by Xi and Pfenning [17] to eliminate array bound checks with a type-based approach. If time admits, we will investigate into dependent and refinement types to explore possibilities of integrating such types in our proposed type system to Octave for array bound and matrix dimension checks.

5 Motivation

Introducing gradual typing to a language is not in itself novel work; as discussed above there are a number of such projects. However, we believe that there is inherently much value in applying this to the domain of numerical computing languages such as Octave or MATLAB; and while some work

has been done already with type and shape inferencing followed by language transformations for performance benefits, there has been little discussion of taking the gradual typing approach.

Hence, the key distinguisher between our proposed project and the related projects above is that we wish to directly apply the notion of gradual typing to Octave. By introducing optional syntactic forms for type specification, the goal is for users to gradually transition away from pure dynamically-typed codebases while also maintaining the flexibility to leave certain pieces untouched. This matter also distinguishes our work from that of Hendren et al. [9], who do not allow for a gradual transition to a statically-typed system.

Previous projects in the same domain such as Chauhan et al. [5] and Olmos et al.'s [12] works have largely abstracted the underlying program transformations away from the user and are largely ignored during development time until compilation; though in the former case annotations are provided to assist inference [5].

Thus, the intent here differs slightly from the projects mentioned. Rather than focusing primarily on performance benefits that can be created through applying static reasoning over provided code, we hope to introduce a new type paradigm for the Octave programming language that is core to the user experience and provides an additional set of tools for the developers that enables them to directly add static type and shape guards. In this sense, our design deviates in that it encourages users to approach development differently and to use static types as much as possible, though we still ensure that existing code is able to run as is.

In fact, there are also performance benefits to adopting a gradual typing approach, in addition to the previous telescoping and program transformation techniques employed above. Recent development of gradually-typed compilers have been successful in attaining performance on par with statically-typed compilers [11]. As a result, the Octave variant that we propose will allow one to perform additional optimizations in the compile phase and therefore have equal or better performance to existing compilers in general.

As a proof of concept for gradually-typed Octave, we first introduce a static-analysis tool for ensuring type and shape consistency within Octave scripts optionally augmented by the new typing syntax. In a fashion similar to TypeScript, we will initially compile the augmented Octave scripts back to the original syntax to be run. However, we approach this with the hope that in the future a compiler that capitalizes on the performance benefits alluded to in the discussion above may be developed as a follow-up project.

6 Project Goals

For the final project milestone, we intend to enrich our proposed gradual type system with more domain-specific features, such as static dimension checking for matrix operations, as matrices are one of the most significant component of the Octave language. We plan to survey and experiment various approaches to enforce type checking for these language-specific features, for which we have seen several design choices. As a potential future extension, we will look into how we can convert the annotated Octave code back into compilable Octave.

7 Conclusion

The objective for this project is to develop a gradually-typed variant for Octave that sufficiently expresses the basic data types of the language. We believe that the flexible nature of gradual typing offers great value to data scientists and various users of Octave in that it remains an effective prototyping tool while optionally providing compile-time type and invariant assertions. To achieve this, we intend to base our initial prototypes for static semantics on the existing literature described above. Additionally – and as time permits – we intend to enrich and further introduce nuances into our gradual typing scheme by delving into features such as static dimension checking (as matrices are a core component of the language domain) and other more domain-specific aspects. We hope to create a meaningful and useful tool for Octave developers to use and ideally contribute towards in the future.

Acknowledgments

The authors would like to thank the course staff of CPSC 311 at University of British Columbia for their feedback.

References

[1] [n. d.]. MATLAB parser by ericharley.
[2] [n. d.]. MATLAB parser by ewiger.
[3] [n. d.]. MATLAB Programming/Differences between Octave and MATLAB. https://en.wikibooks.org/wiki/MATLAB_Programming/Differences_between_Octave_and_MATLAB
[4] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on Object-Oriented Programming*. Springer, 257–281.
[5] Arun Chauhan, Ken Kennedy, and Cheryl McCosh. 2003. *Type-based speculative specialization in a telescoping compiler for MATLAB.* Technical Report.
[6] Ismael Figueroa, Éric Tanter, and Nicolas Tabareau. 2012. A practical monadic aspect weaver. In *Proceedings of the eleventh workshop on Foundations of Aspect-Oriented Languages*. ACM, 21–26.
[7] Michael Furr. 2009. *Combining static and dynamic typing in Ruby.* Ph.D. Dissertation.
[8] Ronald Garcia and Matteo Cimini. 2015. Principal type schemes for gradual programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 303–315.
[9] Laurie Hendren. 2011. Typing aspects for MATLAB. In *Proceedings of the sixth annual workshop on Domain-specific aspect languages*. ACM, 13–18.

[10] Andrew Kent. 2017. Refinement Types in Typed Racket.
[11] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. 2018. Efficient Gradual Typing. *arXiv preprint arXiv:1802.06375* (2018).
[12] Karina Olmos and Eelco Visser. 2003. Turning dynamic typing into static typing by program specialization in a compiler front-end for Octave. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. IEEE, 141–150.
[13] Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.
[14] 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.
[15] 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.
[16] 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.
[17] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Notices*, Vol. 33. ACM, 249–257.