Plan/Proof-of-Concept: 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

TODO: In this proof of concept for our proposed project, we explore advantages of applying gradual typing for Octave and additionally discuss the proposed project in the context of existing work. Finally, we contend the novelty of the project in the context of existing work in the domain as well as provide an overview of the project milestones and approach.

Keywords gradual typing, Octave

1 Introduction

Static and dynamic type systems for languages have their own distinct advantages. For instance, a static type system enables early error detection and enforces a certain extent of code style within a collaborative setting. On the other hand, the lightweight workflow associated with dynamic typing is highly suited for rapid prototyping and iterative approaches. Over the past several decades, researchers in the programming language community have been working on integrating aspects of both static typing and dynamic typing with the goal of allowing programmers access to advantages of both type systems. Gradual typing—originally proposed by Siek and Taha [13]—is one such solution that combines the two type systems and allows the end-users to optionally provide typing information. In recent times, it has largely gained traction in the programming language community and has been adopted by many programming languages both within industry and academia. Certain examples include Typed Racket [15], TypeScript [2] and Reticulated Python [16].

Our proposed project is to introduce gradual typing to Octave. As an interpreted numerical computation language, gradual typing benefits the environment by providing robustness and performance suitable for transition to production environments. Further, we intend to introduce backwards-compatible typing syntax to allow programmers to seamlessly and gradually evolve their programs without refactoring first to other languages. The following report will thus provide brief justification and possible value for such a project, and additionally to understand its place within the larger context of other related work.

2 Advantages of Octave

2.1 Octave Is Relevant

Octave is an open-source scientific programming language that uses a dynamic type system. It is widely employed in the domain of statistics, mathematics and machine learning for idea validation and fast prototyping. Octave shares the vast majority of its syntax and functionality with MATLAB, including but not limited to aspects such as having 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 [1]. Therefore there is already a large audience and domain to potentially engage with for the proposed project. Note that in the context of domain research for the current discussion, we likewise refer to several supporting sources for MATLAB due to the near-analogous properties of the two languages.

2.2 Octave Benefits From Gradual Typing

In a paper describing program specializations used to produce efficient function overloading in Octave [12], Olmos et al. clearly illustrate the benefits of introducing static type checking and shape analysis into Octave. In their work, static type and shape inferencing is used to directly reduce the number of type and shape checks needed at runtime and in doing so they can improve the efficiency of existing code. Functions and operators are highly overloaded in Octave to provide a simple interface for the end-users; however, the inevitable tradeoff for this flexibility is computational performance at runtime. The runtime system is responsible

for type checking, array shape determination, function call dispatching, and handling possible runtime errors. Hence Octave suffers from a number of computational inefficiencies that may be improved provided additional static type and shape context during the compilation step.

2.3 Octave Has Reference Parsers

It is also convenient that Octave itself is open-source and serves as a great point of reference. Specifically, the parser for Octave is available online, while that of MATLAB is proprietary and inaccessible. That being said, there are indeed many open source projects related to MATLAB and 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, which is processed to an abstract syntax tree and control flow graph [8]. 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 an abstract syntax tree [18].

2.4 Octave Type Domain Is Interesting

Octave as a numerical computation language provides an interesting domain to design and implement syntax for static typing as dimensionality or shape also plays a significant role in determining type consistency. Analysis and consistency assertions for shape will be the basis of our long-term goals for the project.

3 Advantages of 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.

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 single piece of code. On the other hand, dynamically-typed languages are usually more flexible and accept various disciplines or coding practices, making them more suitable for tasks like idea validation, fast prototyping and scripting. Static typing, however, enforces certain coding disciplines and some degree of abstraction, which forces programmers to consider these design questions from the very beginning. Therefore, there are sure benefits and pitfalls that exist within 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 while enjoying the flexibility of dynamic typing in other regions, integrating the benefits from both type paradigms 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`):

$$
\begin{aligned}
&\texttt{(}\lambda\ \texttt{(x:?) (succ x)) 1}\\
\longmapsto\ &\texttt{(}\lambda\ \texttt{(x:?) (succ } \langle \texttt{? } \rightarrow \texttt{ int} \rangle\ \texttt{ x)) } \langle \texttt{int } \rightarrow \texttt{ ?} \rangle\ \texttt{ 1}\\
\longmapsto\ &\texttt{succ } \langle \texttt{? } \rightarrow \texttt{ int} \rangle\ \langle \texttt{int } \rightarrow \texttt{ ?} \rangle\ \texttt{ 1}\\
\longmapsto\ &\texttt{succ 1}\\
\longmapsto\ &\texttt{2}
\end{aligned}
$$

Because of explicit casts at runtime, there may be concerns for 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 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 is unable to compete with that of statically-typed languages as static type systems are able to check many aspects of computer programs

well before they execute. In order to resolve the issues mentioned above, prototyping programs in industry are sometimes refactored to statically-typed languages, in order to reduce the possibility of errors as well as to enhance runtime performance.

This process, nevertheless, is undoubtedly time-consuming and error-prone. The gradual guarantee ensures that computer programs remain well-typed as long as type annotations are correctly added [14]. The opposite—removing type annotations from a program does not break programs—is likewise always guaranteed 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 Octave and MATLAB

In this section we explore and discuss existing projects developed for Octave related to typing. Moreover, having noted the significant commonality between MATLAB and Octave, we also extend our analysis of related projects to those developed for MATLAB.

While some work has been done already with type and shape inferencing followed by language transformations for performance benefits, there has been a lack of discussion for applying the gradual typing approach directly.

4.1.1 Optimizing Function Overloading in Octave

Octave and MATLAB were designed to be easy to use and have syntax that follow closely with mathematical notation. However, with dynamic type checking, there are issues such as efficiency and the quality of the generated code. As briefly alluded to before, Olmos and Visser [12] specifically address 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 main disadvantage to this approach is that the resulting Octave syntax is no longer as close to mathematical notation as before. However, by inferring the types used in the program for static typing, the system does not require any additional annotations from the user; hence the user developing their program is not greatly impacted. Since transformed program is statically typed, the overall computational performance of the system is improved. Thus, this paper shows how mixing dynamic and static typing can improve the overall system for Octave.

4.1.2 Typing Aspects in MATLAB

In their paper "Typing Aspects for MATLAB" [9], Hendren explains the process of adding typing aspects to MATLAB via “atype” statements, which are a form of type annotation used to specify runtime types of variables to effectively form type guards 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 the type-annotated MATLAB to the standard MATLAB syntax so that it still compiles normally. However, the runtime checks are implemented as aspects can only execute at runtime, and therefore do not offer additional utility besides providing "... a mechanism for specifying, capturing and checking the dynamic types in MATLAB" [9].

4.1.3 Telescoping Compiler for MATLAB

Another aspect in which the notion of static typing has been explored for MATLAB is through the method of telescoping. Chauhan et al. [3] describe telescoping as "a strategy to automatically generate highly-optimized domain-specific libraries" and in their work suggest a method to create a telescoping system that generates high performance libraries from prototype MATLAB. In order to accomplish this, it needs to infer MATLAB types. The paper develops an approach of combining static and dynamic type inference roughly as follows: to start the inference, constraints are formed using a database of annotations containing one entry per procedure or operations. Unfortunately, the problem of inferring the types is NP-hard. In order to obtain a relatively more efficient algorithm, a few simplifying assumptions were made to reduce the same problem 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 by this method. A key observation that we note from this paper is that typed MATLAB certainly has several benefits but the act of performing 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)

Diamondback Ruby [5] 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 [6].

Although both DRuby and our proposed project 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 static type inference to analyze Ruby programs, it requires static type checking for the entire program. That is, some Ruby programs does not type-check in DRuby. This design, however, violates the criteria for gradual typing, which says gradual type systems must accept both fully untyped programs and fully typed programs [14]. 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 [7] 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

Typed Racket [15] 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 certain type annotations may be omitted by the 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, improving runtime performance of Typed Racket to some extent. However, it also introduces some inconvenience on the programmer side. As reported in [4], 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 were first introduced by Xi and Pfenning [17] to eliminate array bound checks with a type-based approach. If time permits, we intend to additionally investigate the use of dependent and refinement types in our proposed type system to Octave to allow for array bound and matrix dimension checks.

5 Project Distinction and Novel Aspects

There are currently no gradual typing projects for Octave, and from the discussion above we have identified several advantages in applying gradual typing to the language.

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. While Hendren [9] similarly allows for optional annotation, it accomplishes this through an aspect-oriented design which ultimately still executes dynamically and fails to address the matters raised in the motivation for our project.

Additionally, previous projects such as Chauhan et al. [3] and Olmos and Visser [12] have largely abstracted the underlying program specializations away from the user and are largely ignored during development time until compilation. These do not intend to provide alternate workflows for the programmer but instead aim to offer methods to produce optimized code for execution. Thus, the intent here differs slightly from the projects mentioned. Rather than focusing primarily on the 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 specialization techniques employed above. We again cite that 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 for additional optimizations in the compile phase and therefore have at least equal or better performance to existing compilers in general depending on the proportion of statically-typed code.

6 Proposed Project

As the next step to meet the teamâĂŹs full project goals, and in a fashion akin to both TypeScript and typing aspects, we will compile the augmented Octave scripts back to the original syntax to be run. In order to compile the typed Octave code into executable Octave code, we will recursively generate Octave code directly from our high-level AST. We believe that this will be a manageable task as, given instances of our AST representation, there is sufficient information to re-create the original semantics. This is beneficial because not only can a user type check their code, they can also execute the type checked code in their original environment.

However, we approach this with the hope that in the future a compiler that capitalizes on the performance benefits may be developed as a follow-up project; though this is out of scope for the current project.

In addition, we intend to enrich and further introduce nuances into our gradual typing scheme by delving into features such as static dimension checking provided that matrices are a core component of the language domain. Octave can include matrices to do mathematical calculations such as element operations, vector operations, and other ways to manipulate matrices. We believe that this is a challenging task given that we introduce the notion of shapes as an additional constraint to our type checking. Furthermore, there will inherently be many syntactic ambiguities introduced to parsing. One such example is the ambiguity between matrix extraction and regular function call, another is simply the matter that many primitive operators such as arithmetic operators are heavily overloaded in Octave and so correctly capturing the surrounding context is an important matter.

For this task, we now propose the following preliminary syntax for matrices and lists (which we treat as a simple desugaring to a 1 by n matrix):

```
identifierA : matrix[Type][n, m]
identifierB : list[Type]
```

For example, if we had `m:matrix[int][2,2] = [1,2; 3,4]`, this will successfully type check. In contrast, if we have `m:matrix[int][2,2] = [1,2; 3,4; 5,6]`, this will violate the dimension check as we create a 3x2 matrix instead of the declared 2x2. Additionally, `m:matrix[string][2,2] = [1,2; 3,4]` will violate the type consistency of the matrix. Since we are introducing gradual typing, we will also allow `matrix[2,2] = [1,2; 3,4]` which would allow any type to be assigned to the matrix.

We hope to create a meaningful and useful tool for Octave developers to use and ideally contribute towards in the future.

7 Conclusion

In this report we have provided motivation for our proposed project of introducing gradual typing to Octave. We contend that Octave is an ideal candidate language as it is a well-known and relevant language, it benefits from the process of gradual typing and has many reference resources. Additionally, we show that through gradual typing benefits of both static and dynamic typing systems can be incorporated efficiently and to enable program evolution. We further discuss related projects for Octave and MATLAB, as well as gradual typing and claim that our project is unique and is meaningful within the larger context.

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] 2018. MATLAB Programming/Differences between Octave and MATLAB. https://en.wikibooks.org/wiki/MATLAB_Programming/Differences_between_Octave_and_MATLAB

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

[3] Arun Chauhan, Ken Kennedy, and Cheryl McCosh. 2003. *Type-based speculative specialization in a telescoping compiler for MATLAB.* Technical Report.

[4] 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.

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

[6] Michael Furr, Jong hoon (David) An, Mark Daly, Benjamin Kirzhner, Avik Chaudhuri, Jeffrey Foster, and Michael Hicks. 2009. Diamondback Ruby. http://www.cs.umd.edu/projects/PL/druby/

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

[8] Eric Harley. 2013. https://github.com/ericharley/matlab-parser.

[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. http://blog.racket-lang.org/2017/11/adding-refinement-types.html

[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.

[18] Yauhen Yakimovic. 2013. https://github.com/ewiger/decade.