Final Project: 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 full project report 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. We contend the novelty of the project in the context of existing work in the domain and also provide an in-depth tutorial on the full project we have constructed to demonstrate a potential approach. Finally, we provide a plan and overview of the remaining future work that could be added to further extend this project.

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 [8]—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 [10], TypeScript [1] and Reticulated Python [11]. In this report, we extend our current proof of concept to support gradual typing for matrices as well as convert typed Octave back into original compilable Octave source code. Additionally, we propose a few future reach goals that can be completed to further extend this project.

2 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.

Additionally, previous projects such as Chauhan et al. [2] and Olmos and Visser [6] 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 [4]. As a result, the Octave variant that we propose will be 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.

3 Matrix Type Checking

3.1 Parsing

For parsing, we have updated the BNF file to handle the new syntax and have introduced a new matrixT AST type that can be parameterized with shape and type information.

The updated language file can be found here: https://github.com/yuchong-pan/gradual-octave/blob/master/proof-of-concept/language.rkt

A point to note is that we treat matrix index operations the same as we do with function calls initially. For example, the following expression `X = [1 2; 3 4]; X(1,1)` currently is treated exactly as a function application, though we are instead retrieving the element at index `(1,1)`. In fact, we make no such distinction within the BNF itself between a function application and an indexing operation as they share the same syntactic form.

We can resolve this by performing a lookup in our desugaring step and introduce a new conjoined struct `i-app-index` similar to the use of the `i-assn-decl struct`. As we desugar, we simply check what is currently in scope for the definition of `X`; if `X` is a matrix, then we perform an index operation—and if `X` is a function, we create a function application. In fact, the index operation itself is a desugaring for a function application, as the input and return type are given by the input matrix and the indexing arguments.

3.1.1 Type checking

To type check matrices, we first need to adapt the type consistency rules for the newly added matrix types. This, however, is very similar to the arrow type which we already accomplished during the proof-of-concept stage, because both the arrow type and the matrix type have type parameters. In addition to that, the matrix type also includes value parameters as its dimensions. Hence, two matrix types are consistent if their dimensions match and their type parameters are the same. With the updated type consistency rules, most language constructs automatically work for the matrix type. The new type consistency checking helper function is given in Figure 1.

```
(define (consistent? T1 T2)
  (match `(,T1 ,T2)
    ;; ...
    [`(,(matrix element-type1 n1 m1)
       ,(matrix element-type2 n2 m2))
     (and (consistent? element-type1 element-type2)
          (= n1 n2)
          (= m1 m2))]))
```

Figure 1. Helper function to check type consistency

As noted above, we have modified our abstract syntax tree structure to make it more suitable for the same operation on different types. Therefore, we merely need to perform case analysis when type-checking each operation. For instance, for the `*` operation, we have cases where both operands are integers/strings/Booleans and where both operands are matrices. In the latter case, we then need to perform additional type checking to determine if the dimensions of the two operands match; i.e., we will check if the types of the two operands are of the form `matrix[T][n, m]` and `matrix[T][m, p]`, where `n`, `m` and `p` are constant integers and `T` is a type parameter. The type checker for the `*` operation is illustrated by the code snippet in Figure 2. Here, the function meet illustrated in Figure 3 computes the meet of the two types, i.e., the greater lower bound of the two types, where the relation is the "less or equally dynamic" relation [7].

```
(define (typecheck-op* e)
  (local [(define lhs (op*-lhs e))
          (define rhs (op*-rhs e))
          (define ltype (typecheck-expr lhs))
          (define rtype (typecheck-expr rhs))]
      (match `(,ltype ,rtype)
        [`(int int) int]
        ;; ...
        [`(,(matrix type1 n1 m1)
           ,(matrix type2 n2 m2))
           (if (and (equal? type1 type2)
                    (= m1 n2))
               (matrix (meet type1 type2)
                       n1
                       m2)
               (error 'typecheck-op* "..."))])))
```

Figure 2. Helper function to type-check the multiplication operation

4 Code Generation

4.1 Design

The codegen module in essence is a traversal over the AST that recursively converts components of the AST into strings and appends them, all while maintaining depth information for indentation. For instance, statements within a function body or if statement will have their depths increased by one and so forth. While it was not required for us to properly space the code, we believe that doing so allows for enhanced readability and clarity in the generated code—thus allowing a user to transparently see the transformations that are applied to the original Gradual Octave source.

4.2 Modifications to AST

As we were previously designing the AST, we did not account for code generation; thus, did not maintain a set of operator

```
(define (meet T1 T2)
  (match `(,T1 ,T2)
    [`(T1 dynamic) T1]
    [`(dynamic T2) T2]
    [`(,(arrow T1-dom T1-cod)
       ,(arrow T2-dom T2-cod))
     (arrow (map meet T1-dom T2-dom)
                (map meet T1-cod T2-cod))]
    [`(,(matrix element-type1 n1 m1)
       ,(matrix element-type2 n2 m2))
     (and (equal? element-type1 element-type2)
          (= n1 n2)
          (= m1 m2))]
    [(equal? T1 T2) T1]
    [else (error 'meet "...")]))
```

Figure 3. Helper function to compute the meet of two types

symbols for internal arithmetic and logical expressions. The functional information was only represented in an abstract manner using lambda functions. However, the updated AST for matrices described in the previous section allowed us to have sufficient context to generate the original operators being used.

4.3 Examples

The following are some examples of Gradual Octave source with their corresponding generated Octave using our code generation module:

As depicted above, the outputs strip typing information from the typed syntax so that it is compatible with the original Octave interpreter.

5 Accomplishments

The goal of this project was to introduce gradual typing to dynamically typed Octave. We started by taking in Octave source code and parsing that into our internal AST. We also wrote a type checker for our AST in our proof of concept which supports types such as integers, boolean, strings, dynamic, void, functions, and star types. The four main stages of parsing include 1) tokenization 2) parsing to a Racket syntax object 3) parsing the object to an intermediate syntax 4) desugaring into our final abstract syntax tree. For our final project, we further extend the types we support to include matrices as they are widely used in mathematical and statistical calculations. Lastly, we created a code generation program to convert our internal AST back into compilable Octave source code. With these accomplishments, we have introduced gradual typing for a subset of Octave while ensuring users can conveniently use this without having to rewrite their Octave source code. We also have a few future goals for this project that we will elaborate in the section below.

6 Future Goals

To further extend this project, we propose a few future goals that can enhance the capabilities of this project. Potential high risk reach goals are: web IDE, dependent type support, and performance benchmark.

6.1 Web IDE

First, we can implement a web IDE that helps with debugging code with a GUI component. The main motive behind this is we want to be able to have static type checks running constantly in the background. This will enhance usability as the GUI will be able to constantly display typing errors instead of requiring the user to periodically ask for feedback. We believe usability is a big factor in drawing in users to a new tool. with a GUI, it will create an easier learning experience as well as a cleaner environment. The editor we have in mind will be similar to IDEs such as Visual Studio Code or IntelliJ.

Moreover, we can support type inference in our gradual extension to Octave, in order to automatically type-check untyped Octave programs. This will be very useful for a gradual migration from untyped code to fully-typed programs, and integrating type inference in the web IDE will largely aid programmers in inferring types from untyped Octave code and Octave libraries. Siek and Vachharajani [9] discussed the integration of gradual typing and type inference, and proposed a unification-based algorithm to perform type inference in a gradual type system. Garcia and Cimini [3] proposed a new approach to infer types in a gradually-typed language.

6.2 Dependent Type Support

Second, we can develop a dependent type support for boundary checks. The idea was first proposed by Xi and Pfenning [12], and our syntax design is similar to the syntax provided in that paper. Moreover, interoperability between gradual typing and dependent types was discussed by Lehmann and Tanter [5].

Currently, with our matrix type checking, we only support static dimension checks, i.e., the dimensions of a matrix type can only be constant integers such as `matrix[int][2,3]`. With dependent type support, we can introduce type checking for variable matrix dimensions. An example of what this can support is shown in Figure 4.

```
{n:int, m:int, p:int}
function C:matrix[int][n,p] =
        f(A:matrix[int][n,m], B:matrix[int][m,p])
    # ...
endfunction
```

Figure 4. Matrix arguments with parametric dimensions

With this, we can both allow the user to type check dimensions while also giving the user flexibility on what the dimension values will be. Users can also mix in static dimensions with dynamic dimensions. In that case, we will propose a syntax shown in Figure 5.

```
{m:int, p:int}
function C:matrix[int][5,p] =
        f(A:matrix[int][5,m], B:matrix[int][m,p])
    # ...
endfunction
```

Figure 5. Matrix arguments with parametric and static dimensions

Additionally, we can add conditions in dependent types to further restrict the relations amongst parameters. We will propose a syntax like `{i:int|i < n}` to represent a subset type of `{i:int}` with additional constraints that the variable i is strictly less than the variable n. With this syntax, we can easily eliminate matrix boundary checks in a type-based approach, instead of checking them at runtime. We anticipate that this extension would largely enhance the performance of Octave programs because time-consuming runtime boundary checks are eliminated.

By introducing dependent type support, we believe it will give programmers more flexibility while still ensuring type checking is in place.

6.3 Performance Benchmark

Finally, we can compare the performance of the compiled version versus the generated code in untyped Octave. Currently, we generate the untyped Octave code from our internal AST as we believe that is the fastest way given we don't have an optimized compiler for Octave. If this project was extended, it would be ideal to have a compiler implemented. However, a compiler with no optimizations could end up being even slower than generating original code as complex programs would suffer from the slow runtime. In contrast, an optimized compiler will be non-trivial but we believe it will be faster than the original compiler for Octave source code in certain cases. With an optimized compiler, it can make use of the annotated types to do static type checks. This will allow the static regions to compile faster than the original compiler. For dynamic regions, it will either be similar in runtime or a little bit slower. This is due to the fact that the compiler will still have to do runtime checks and possibly have to do type casts with the annotated types added in. We believe this would be an interesting case to investigate to determine how beneficial it is to have an optimized compiler written for this project given it's difficulty.

7 Concluding Remarks

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 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 initially based our prototypes for static semantics on the existing literature described above. Additionally, we now 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 code generation. We hope we have created a meaningful and useful tool for Octave developers to use and ideally be able to further contribute towards this project's reach goals.

Acknowledgments

The authors would like to thank the course staff of CPSC 311 at University of British Columbia for their feedback. In particular, the authors would like to thank Steve Wolfman for the opportunity to create this project, and to thank the project coordinator Sam Chow and other anonymous graders for their suggestions for improvement of this project.

References

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

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

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

[4] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. 2018. Efficient Gradual Typing. *arXiv preprint arXiv:1802.06375* (2018).

[5] Nico Lehmann and Éric Tanter. 2017. Gradual refinement types. *ACM SIGPLAN Notices* 52, 1 (2017), 775–788.

[6] 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.

[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 and Manish Vachharajani. 2008. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM, 7.

[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.

[12] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Notices*, Vol. 33. ACM, 249–257.