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

In this report on the plan and 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. We contend the novelty of the project in the context of existing work in the domain and also provide an in-depth tutorial on a proof-of-concept we have constructed to demonstrate a potential approach. Finally, we provide a plan and overview of the remaining work for the 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 [12]—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 [3] and Reticulated Python [16].

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. [4] and Olmos and Visser [11] 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 [9]. 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.

3 Proof-of-Concept

For the full proof-of-concept, see the project repository at https://github.com/yuchong-pan/cpsc-311-project. Below, we provide a tutorial for building the pipeline used to type-check our proposed gradually-typed Octave source files. To demonstrate the entire process of conversion from source code to a check result, we will show various stages of output as it is processed through our application. Figure 1 illustrates an overview of the final structure of our proposed project.

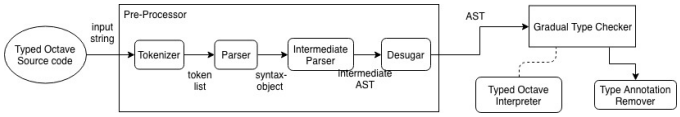

Figure 1. Overview of Our Proposed Project

3.1 Pre-processing

Before we begin static analysis of type information for a source file, we must convert it to our abstract representation. 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. As the first two steps are largely coupled, we address them together in a single section below.

3.1.1 Tokenization and Parsing

For this proof of concept, we decided to perform our own lexical analysis and parsing using modified syntax that defines a subset of Octave with type information. However, we note that future work could be done to extend the official Octave open source parser [5] to handle this stage for completeness. Figure 2 includes examples of rules used in our BNF.

We begin this stage by performing both lexical analysis and parsing using tools from the Racket brag library [18]. In the first step of pre-processing, we load Octave source code into memory as a string and perform lexical analysis using a lexer function. For this, we have defined tokens for each of the terminals defined in the language such as for identifiers, constants, as well as literal characters.

The result of passing the source code string to this method is a generator function that, when called, generates a token list such as the one pictured in Figure 3.

Using the brag module, we obtain a parse function from the language file shown above which takes in a token list (or generator function) and ultimately constructs a syntax object [6] which provides an easily traversable form for us to desugar. The syntax in Figure 4 form is an abstract representation of the input source file in terms of the defined BNF.

3.1.2 Intermediate Representation

Now that we have a structure that can be easily traversed by Racket, we are ready to transform the parsed syntax entity to an intermediate abstract syntax tree. Below we provide brief example structures that are used for this syntax, though we elaborate more on these in the main section to follow on type-checking.

```
octave
        : translation_unit
        | octave translation_unit
        ;

translation_unit
        : statement_list
        | FUNCTION function_declare eostmt
          statement_list eostmt ENDFUNCTION
          eostmt
        ;

primary_expression
        : typed_identifier
        | BOOLEAN
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        | '[' ']'
        | '[' array_list ']'
        ;
```

Figure 2. Selected BNF rules

```
(list
 (token-struct 'IDENTIFIER "a" #f #f #f #f #f)
 (token-struct 'WHITESPACE " " #f #f #f #f #t)
 (token-struct '= "=" #f #f #f #f #f)
 (token-struct 'WHITESPACE " " #f #f #f #f #t)
 (token-struct 'CONSTANT "3" #f #f #f #f #f))
```

Figure 3. Sample token list

```
'(octave
  (octave
   (translation_unit
    "function"
    (function_declare
     (func_return_list
      "["
      (func_ident_list
       (func_ident_list (typed_identifier "outx"))
       ","
       (typed_identifier "outy"))
      "]")
     "="
     ;; ...
```

Figure 4. Sample syntax structure

To generate the abstract syntax tree, we use the PLAI Scheme language [1] to match on the syntax structures produced by the previous step. To do this, we split each syntax object into a list using the syntax->list operation and this effectively allows us to parse 1) the current rule 2) the arguments for the rule.

We then extract the meaningful information captured within each rule, and produce a list of statement or function entities that describe the static form of the program. This includes any typing information that was declared within the source code. Figure 5 is the example intermediate syntax obtained from this step.

```
(list
 (i-func
  (i-id-type 'f 'dynamic)
  (list (i-id-type 'y 'dynamic)
        (i-id-type 'x 'dynamic))
  (list (i-id-type 'outy 'dynamic)
        (i-id-type 'outx 'dynamic))
  (list
   (i-assn-decl (list (i-id-type 'outx 'dynamic))
                      (i-id-type 'y 'dynamic))
   (i-assn-decl (list (i-id-type 'outy 'dynamic))
                      (i-id-type 'x 'dynamic))))
 (i-assn-decl
  (list (i-id-type 'x 'dynamic))
  (i-app (i-id-type 'f 'dynamic)
         (list (int 1) (int 2)))))
```

Figure 5. Intermediate syntax for source code

3.1.3 Desugaring

Finally, we must desugar the intermediate syntax into the final abstract syntax. In this stage, one major goal is to insert declarations for variable assignments as they are being assigned to. We do this by scoping variables to environments and performing lookups to find "unbound" variables that are being assigned to. Once bound, the type of the variable is fixed from that point onwards. Additionally, in the process of converting to the abstract syntax we strip off unnecessary type information that can be bundled with identifiers.

At this point, the original source file has been transformed into our desired form and we can perform type-checking on it. In the section to follow, we will elaborate on the form and use of our abstract syntax.

3.2 Type-Checking

3.2.1 Type Consistency

One can view dynamic typing as a type system with only one type `dynamic`. Thus, the main difference of a gradual type system from a static type system is the additional unknown type, denoted by ?, which is used to indicate a partially-known structure of a type [12]. For instance, `int` $\rightarrow$ `?` represents a function type whose domain is `int` and whose co-domain can be any type.

In order to support the unknown type, the type equality relation is no longer working in a gradual type system. Consider a function `f` with type `int` $\rightarrow$ `int`, and a variable `x` with type ?. A gradual type system should allow `f` to be applied on `x` by casting x from type ? to type `int` in the runtime, whereas type ? is not equal to type `int`. Hence, an important difference of a gradual type system froma static type system is to replace the type equality relation with the type consistency relation [12], which is defined in Figure fig:consistency. Here, the arrow type (`→ T T`) is an explicit function type, and the star type `T` $\times$ `T` is the type for pairs of two instances of difference types.

$$\begin{array}{lll} B & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{string} \\ T & ::= & B \mid (\rightarrow\ \texttt{T}\ \texttt{T}) \mid \texttt{T} \times \texttt{T} \mid \texttt{?} \mid \texttt{none} \end{array}$$

Figure 6. Type definitions

$$\frac{}{\texttt{?} \sim T} \qquad \frac{}{T \sim \texttt{?}} \qquad \frac{}{B \sim B}$$

$$\frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \rightarrow T_2 \sim T_3 \rightarrow T_4} \qquad \frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \times T_2 \sim T_3 \times T_4}$$

Figure 7. Type consistency rules

The type consistency relation needs to respect several properties. First, type consistency should *conservatively* extend the type equality relation [8]. Described as a criterion for gradual typing, a gradual type system should behave exactly the same as a static type system for fully statically-typed programs [14]. Hence, two basic types T_1 and T_2 are consistent if and only if they are equal. Second, it is easy to see that type consistency should be reflexive, symmetric, but not transitive. To see the non-transitivity, if consistency were transitive, then `int` $\sim$ `?` and `?` $\sim$ `bool` would imply that `int` $\sim$ `bool`, which would violate the criterion for gradual typing stated above. In our type-checker, we implemented the helper function `consistent?` to check type consistency of two types, given in Figure 8.

3.2.2 Typing Rules

Figure 6 is the typing rules that we follow in the proof-of-concept stage. It extends the type system of the gradually-typed lambda calculus (GTLC) [12]. The typing rules described in Figure 6 can be read as "if the propositions above the bar are true, then the proposition below the bar will be

```
(define (consistent? T1 T2)
  (match `(,T1 ,T2)
    [`(,T1 dynamic) #t]
    [`(dynamic ,T2) #t]
    [`(,(arrow T1-dom T1-cod)
       ,(arrow T2-dom T2-cod))
     (and (consistent-list? T1-dom T2-dom)
          (consistent-list? T1-cod T2-cod))]
    [else (equal? T1 T2)]))
```

Figure 8. Helper function to check type consistency

true", and the type judgements "$\Gamma \vdash e : T$" can be read as "expression e has type T in the context Γ". The metafunction fun extracts the domain and co-domain from a possible function type (i.e., the ? type can be viewed as a function type $? \rightarrow ?$).

The main extension of our type system to the GTLC type system is the addition of statements because Octave is an imperative language. Statements include assignments and if-statements and do not have types. We represent typing rules for assignments and for if-statements in Figure 10. An assignment type-checks if the type of the left-hand side variable and the type of the right-hand side expression are consistent. An if-statement type-checks if the condition expression has the `bool` type.

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad fun(T_1) = T_{11} \rightarrow T_{12} \quad T_2 \sim T_{11}}{\Gamma \vdash e_1(e_2) : T_{12}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\cdots$$

$$\begin{aligned} fun(T_1 \rightarrow T_2) &= T_1 \rightarrow T_2 \\ fun(?) &= ? \rightarrow ? \end{aligned}$$

Figure 9. Expression typing rules

$$\frac{\Gamma \vdash x : T_1 \quad \Gamma \vdash e : T_2 \quad T_1 \sim T_2}{x \texttt{ := } e}$$

$$\frac{\Gamma \vdash e : \texttt{bool}}{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ endif}}$$

Figure 10. Statement typing rules

3.2.3 Abstract Syntax Tree (AST)

In our abstract syntax tree (AST), statements will be defined as all of the statements we want to support in type-checking, such as expressions, declarations, assignments, function definitions, and if-statements. Note that expressions are always associated with types, whereas statements that are not expressions do not have types.

An important AST node is the function definitions. Unlike many other programming languages, Octave requires us to specify both the identifiers for arguments and those for return variables. Therefore, in our proposed gradual typing extension to Octave, that means both the input and output parameters can be type annotated. To represent that, we use a pair of a symbol and a Type type to represent each argument and each return variable and store them in two lists. The body of the function may also need to be type-checked. We can achieve that by creating a list of statements to represent the body. This way, we can recursively call the type-checker for statement to type-check the body of a function.

```
(struct func
  ([name : Symbol]
   [args : (Listof (Pair Symbol Type))]
   [rets : (Listof (Pair Symbol Type))]
   [body : (Listof Stmt)]))
```

Figure 11. AST node for `func`

Now that we have an idea of how to represent Octave programs as an AST, we can use pass this AST into our type-checker to check for type consistency.

3.2.4 Type-Checker Implementation Details

Type Environments. When a variable or a function is declared, we allow users to specify the type of the variable or the function. However, we need to be able to retrieve this information when we type-check statements and expressions. In order to keep track of what the types of the variables are, we introduce an environment that stores this data. An environment can be represented as `(define-type Env (Listof (Pair Symbol Type)))`. Everytime we encounter a declaration or a function definition, we can append a pair of an identifier and its corresponding type to the environment. Now, we can thread the environment through our program so it can be accessed by statements and expressions. When we get to type-checking a statement or an expression, we can look up the variable name in the environment to retrieve its type. With this environment passed around, we can complete the type-checker.

Statements and Expressions. Our type-checker is broken down into two helper functions to type-check statements and expressions, respectively. Note that the type of an expression will be used to type-check the super-expression

that contains it. For instance, to typecheck the expression $e1$ `+` $e2$, we need to ensure that $e1$ and $e2$ both have type `int`. Therefore, the type-checker needs to type-check each sub-expression and returns the type information of the expression.

In contrast, non-expression statements cannot be further operated on and therefore do not have types. However, it is worth noting that two types of statements, namely declarations and function definitions, will update the type environment; i.e., they will bind new identifiers to their corresponding types in the environment. Therefore, the type-checker for statements returns the possibly updated type environment.

Case Study: Assignments. We will look at one of the more interesting cases such as type-checking an assignment, because Octave allows multiple variables at the left-hand side of an assignment. The type-checker implementation for assignments is given in Figure 12. With an assignment, we first need to determine the types of the left-hand side variables. To do that, we create a list of expected types by searching in the environment and throwing an error if any of the variables are not found. If an error occurs, that would mean this assignment was invalid since we have never declared this variable before. Then, we need to get the actual types. Since the right-hand side of an assignment is an expression, we can call our helper function `typecheck-expr` to type-check the right-hand side.

Note that Octave allows the left-hand side of an assignment to have less variables than the number of values on the right side of the assignment, in which case Octave ignores the last several output values that are missing on the left-hand side. We model this in our type-checker by comparing the length of both sides of the assignment and removing the unnecessary output types from the function application if the length on the right is greater than the length on the left. After this, we also need to make sure the left-hand side of the assignment does not have more variables than the right side of the assignment. For example, we can't have `x, y = 1`. In this case, we will throw an error. If all of the length checks pass, we can simply call our consistent function on the list of types which will check for type consistency between the left-hand side and the right-hand side of the assignment.

Case Study: Function Applications. Function applications are another interesting case that is worth studying. The type-checker implementation for assignments is given in Figure 13. Unlike some other languages, Octave allows the number of actual arguments of a functional to be different from the number of arguments declared in the argument list, and it allows functions to return multiple return values. Handling arguments of function applications is similar to handling assignments, as described above. Therefore, we need to truncate the actual argument list if necessary, and append dummy dynamic types to type-check missing arguments. Then, we check the type consistency between the

```
[(assn names expr)
 (local
   [(define expect-types
      ((inst map Type Symbol)
       (lambda (name)
         (local [(define result (assoc name env))]
           (if (false? result)
               (error 'typecheck-stmt "...")
               (cdr result))))
       names))
    (define expect-length (length expect-types))
    (define actual-types
      (local
        [(define original
           (typecheck-expr env expr))
         (define listify
           (if (list? original)
               original
               (list original)))]
        (if (> (length listify) expect-length)
            (take listify expect-length)
            listify)))
    (define actual-length (length actual-types))]
  (cond
    [(not (= expect-length actual-length))
     (error 'typecheck-stmt "...")]
    [(consistent-list? expect-types actual-types)
     env]
    [else
     (error 'typecheck-stmt "...")]))]
```

Figure 12. Type-checking case for assignments

actual argument list and the expected argument list using the type consistency rule for star types.

It is also worth noting that we may apply a function with the ? type because the ? type can also represent a function type. This may happen when we apply a higher-order function passed in as an argument of another function, where the type annotation of the argument is `dynamic`. To type-check this case, we need to extract the domain type and the co-domain type of the `dynamic` type, which are both `dynamic` according to the metafunction fun is the type consistency rules in Figure 7. Therefore, we implemented two helper functions, `fun-dom` and `fun-cod` to extract the domain and the co-domain of a possible function type.

The rest of the type-checking cases are very similar and simpler than the two cases we described above. They can be implemented by calling the helper functions for statements and expressions as well as calling the consistency function to check the consistency of two types.

4 Plan for Next Steps

As the next step to meet the team's full project goals, and in a fashion akin to both TypeScript and typing aspects, we

```
[(app fun args)
 (local [(define fun-type (typecheck-expr env fun))]
   (if (or (arrow? fun-type)
           (equal? fun-type 'dynamic))
       (local
         [(define expect-types
            (fun-dom fun-type))
          (define actual-types
            (typecheck-list env args))
          (define expect-length
            (length expect-types))
          (define actual-length
            (length actual-types))
          (define adjusted-actual
            (cond
              [(> actual-length expect-length)
               (take actual-types expect-length)]
              [(< actual-length expect-length)
               (append
                actual-types
                (build-list (- expect-length
                               actual-length)
                            (lambda (x)
                              'dynamic)))]
              [else actual-types]))
           (define return-types
           (fun-cod fun-type))]
         (if (consistent-list? expect-types
                               adjusted-actual)
             return-types
             (error 'typecheck-expr "...")))
       (error 'typecheck-expr "...")))])

(: fun-dom (-> Type (Listof Type)))
(define (fun-dom fun-type)
  (cond
    [(arrow? fun-type) (arrow-dom fun-type)]
    [(equal? fun-type 'dynamic) '(dynamic)]
    [else (error 'fun-dom "...")]))

(: fun-cod (-> Type (Listof Type)))
(define (fun-cod fun-type)
  (cond
    [(arrow? fun-type) (arrow-cod fun-type)]
    [(equal? fun-type 'dynamic) '(dynamic)]
    [else (error 'fun-cod "...")]))
```

Figure 13. Type-checking case for function applications

plan on having two low risk core goals as well as three high risk reach goals.

4.1 Low Risk Core Goals

As our plausible simplest suitable approach to meet the team's core project goals, 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.

4.2 High Risk Reach 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.

4.2.1 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 [13] 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 [7] proposed a new approach to infer types in a gradually-typed language.

4.2.2 Dependent Type Support

Second, we can develop a dependent type support for boundary checks. The idea was first proposed by Xi and Pfenning [17], 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 [10].

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 14.

```
{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 14. 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 15.

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

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

Figure 15. Matrix arguments with parametric and static dimensions

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.

4.2.3 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.

5 Concluding Remarks

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. In this proof of concept we provided an overview of the the stages of work we've implemented for the pre-processing as well as type-checking stages. For each stage, we elaborate on the respective tools used and steps taken to achieve certain inputs and outputs. We've also included snippets of the source code to reinforce users' understanding of how to recreate the proof-of-concept without serious difficulties in a tutorial-like format.

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.

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.]. PLAI Scheme. https://docs.racket-lang.org/plai/plai-scheme.html?q=plai

[2] 2018. MATLAB Programming/Differences between Octave and MATLAB. https://en.wikibooks.org/wiki/MATLAB_Programming/Differences_between_Octave_and_MATLAB

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

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

[5] John W. Eaton. 2018. http://hg.savannah.gnu.org/hgweb/octave/file/37e3aa267374/libinterp/parse-tree.

[6] Matthew Flatt, Robert Bruce Findler, and PLT. [n. d.]. Syntax Objects. https://docs.racket-lang.org/guide/stx-obj.html

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

[8] Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 429–442.

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

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

[11] 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.

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

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

[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] Danny Yoo and Matthew Butterick. [n. d.]. brag: a better Racket AST generator. http://docs.racket-lang.org/brag/