

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

Kathy Wang

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

Yuchong Pan

University of British Columbia
yuchong.pan@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 [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 [3] and Reticulated Python [16].

*Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

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 [2]. 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.

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 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 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.

4.2 Diamondback Ruby (DRuby)

Diamondback Ruby [7] 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 [8].

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 [9] 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.

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.

Additionally, previous projects such as Chauhan et al. [4] 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 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.

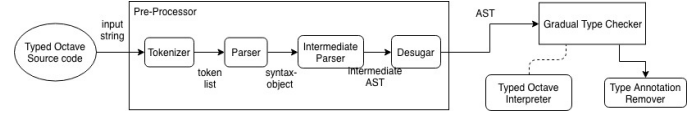


Figure 1. Overview of Our Proposed Project

6.1 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.

6.1.1 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.

```

1 octave
2   : translation_unit
3   | octave translation_unit
4   ;
5
6 translation_unit
7   : statement_list
8   | FUNCTION function_declare eostmt
9     statement_list eostmt ENDFUNCTION
10    eostmt
11   ;
12
13 primary_expression
14   : typed_identifier
15   | BOOLEAN
16   | CONSTANT
17   | STRING_LITERAL
18   | '(' expression ')'
19   | '[' ']'
20   | '[' array_list ']'
21   ;

```

Figure 2. Selected BNF rules

We begin this stage by performing both lexical analysis and parsing using tools from the Racket brag library [17]. 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.

```
1 (list
2  (token-struct 'IDENTIFIER "a" #f #f #f #f #f)
3  (token-struct 'WHITESPACE " " #f #f #f #f #f)
4  (token-struct '= "=" #f #f #f #f #f)
5  (token-struct 'WHITESPACE " " #f #f #f #f #f)
6  (token-struct 'CONSTANT "3" #f #f #f #f #f))
```

Figure 3. Sample token list

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.

```
1 '(octave
2   (octave
3     (translation_unit
4       "function"
5       (function_declare
6         (func_return_list
7           "["
8           (func_ident_list
9             (func_ident_list (typed_identifier "outx"))
10            ", "
11            (typed_identifier "outy"))
12           "]"")
13         "="
14         ;; ...
```

Figure 4. Sample syntax structure

6.1.2 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.

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.

```
1 (list
2  (i-func
3    (i-id-type 'f 'dynamic)
4    (list (i-id-type 'y 'dynamic)
5          (i-id-type 'x 'dynamic))
6    (list (i-id-type 'outy 'dynamic)
7          (i-id-type 'outx 'dynamic))
8    (list
9      (i-assn-decl (list (i-id-type 'outx 'dynamic)
10                        (i-id-type 'y 'dynamic))
11      (i-assn-decl (list (i-id-type 'outy 'dynamic)
12                        (i-id-type 'x 'dynamic))))
13    (i-assn-decl
14      (list (i-id-type 'x 'dynamic))
15      (i-app (i-id-type 'f 'dynamic)
16              (list (int 1) (int 2)))))
```

Figure 5. Intermediate syntax for source code

6.1.3 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.

6.2 Type-Checking

6.2.1 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 [13]. For instance, `int → ?` 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 → 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 from a static type system is to replace the type equality relation with the type consistency relation [13], which is defined in Figure 6:consistency. Here, the arrow type $(\rightarrow T \ T)$ is an explicit function type, and the star type $T \times T$ is the type for pairs of two instances of difference types.

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

Figure 6. Type definitions

$$\begin{array}{c} \overline{? \sim T} \quad \overline{T \sim ?} \quad \overline{B \sim B} \\[10pt] \frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \rightarrow T_2 \sim T_3 \rightarrow T_4} \quad \frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \times T_2 \sim T_3 \times T_4} \end{array}$$

Figure 7. Type consistency rules

The type consistency relation needs to respect several properties. First, type consistency should *conservatively* extend the type equality relation [10]. 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 $\text{int} \sim ?$ and $? \sim \text{bool}$ would imply that $\text{int} \sim \text{bool}$, 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.

```
1 (define (consistent? T1 T2)
2   (match `(. T1 T2)
3     [ `(,T1 dynamic) #t]
4     [ `(dynamic ,T2) #t]
5     [ `(. (arrow T1-dom T1-cod)
6           , (arrow T2-dom T2-cod))
7       (and (consistent-list? T1-dom T2-dom)
8             (consistent-list? T1-cod T2-cod))]
9     [else (equal? T1 T2)]))
```

Figure 8. Helper function to check type consistency

6.2.2 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) [13]. 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 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.

$$\begin{array}{c} \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad \text{fun}(T_1) = T_{11} \rightarrow T_{12} \quad T_2 \sim T_{11}}{\Gamma \vdash e_1(e_2) : T_{12}} \\[10pt] \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \\[10pt] \dots \\[10pt] \begin{array}{lcl} \text{fun}(T_1 \rightarrow T_2) & = & T_1 \rightarrow T_2 \\ \text{fun}(?) & = & ? \rightarrow ? \end{array} \end{array}$$

Figure 9. Expression typing rules

$$\begin{array}{c} \frac{\Gamma \vdash x : T_1 \quad \Gamma \vdash e : T_2 \quad T_1 \sim T_2}{x := e} \\[10pt] \frac{\Gamma \vdash e : \text{bool}}{\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ endif}} \end{array}$$

Figure 10. Statement typing rules

6.2.3 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.

```

1 (struct func
2   ([name : Symbol]
3    [args : (Listof (Pair Symbol Type))]
4    [rets : (Listof (Pair Symbol Type))]
5    [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.

6.2.4 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 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

```

1 [(assn names expr)
2  (local
3    [(define expect-types
4      ((inst map Type Symbol)
5       (lambda (name)
6         (local [(define result (assoc name env))]
7           (if (false? result)
8               (error 'typecheck-stmt "...")
9               (cdr result))))
10     names))
11   (define expect-length (length expect-types))
12   (define actual-types
13     (local
14       [(define original
15         (typecheck-expr env expr))
16        (define listify
17          (if (list? original)
18              original
19              (list original)))]
20       (if (> (length listify) expect-length)
21           (take listify expect-length)
22           listify)))
23   (define actual-length (length actual-types))]
24 (cond
25   [(not (= expect-length actual-length))
26    (error 'typecheck-stmt "...")]
27   [(consistent-list? expect-types actual-types)
28    env]
29   [else
30    (error 'typecheck-stmt "...")]))]

```

Figure 12. Type-checking case for assignments

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.

7 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 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

```

1 [(app fun args)
2  (local [(define fun-type (typecheck-expr env fun))]
3    (if (or (arrow? fun-type)
4            (equal? fun-type 'dynamic))
5        (local
6          [(define expect-types
7            (fun-dom fun-type))
8           (define actual-types
9            (typecheck-list env args))
10          (define expect-length
11            (length expect-types))
12          (define actual-length
13            (length actual-types))
14          (define adjusted-actual
15            (cond
16              [(> actual-length expect-length)
17               (take actual-types expect-length)]
18              [(< actual-length expect-length)
19               (append
20                actual-types
21                (build-list (- expect-length
22                               actual-length)
23                             (lambda (x)
24                               'dynamic)))]
25              [else actual-types]))
26          (define return-types
27            (fun-cod fun-type))
28          (if (consistent-list? expect-types
29                                adjusted-actual)
30              return-types
31              (error 'typecheck-expr "..."))
32          (error 'typecheck-expr "...")))]
33
34 (: fun-dom (-> Type (Listof Type)))
35 (define (fun-dom fun-type)
36   (cond
37     [(arrow? fun-type) (arrow-dom fun-type)]
38     [(equal? fun-type 'dynamic) '(dynamic)]
39     [else (error 'fun-dom "...")])
40
41 (: fun-cod (-> Type (Listof Type)))
42 (define (fun-cod fun-type)
43   (cond
44     [(arrow? fun-type) (arrow-cod fun-type)]
45     [(equal? fun-type 'dynamic) '(dynamic)]
46     [else (error 'fun-cod "...")])

```

Figure 13. Type-checking case for function applications

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.

8 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 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] Michael Furr. 2009. *Combining static and dynamic typing in Ruby*. Ph.D. Dissertation.
- [8] 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/>
- [9] Ronald Garcia and Matteo Cimini. 2015. Principal type schemes for gradual programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 303–315.
- [10] Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 429–442.
- [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] Danny Yoo and Matthew Butterick. [n. d.]. brag: a better Racket AST generator. <http://docs.racket-lang.org/brag/>